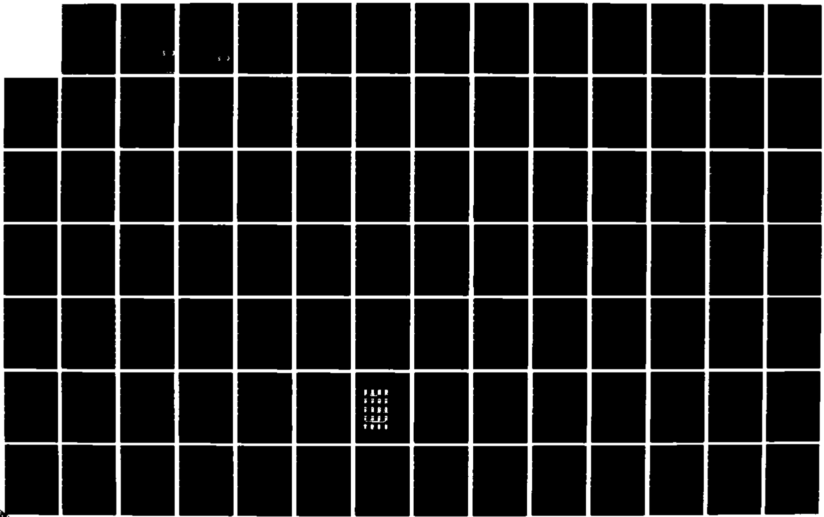
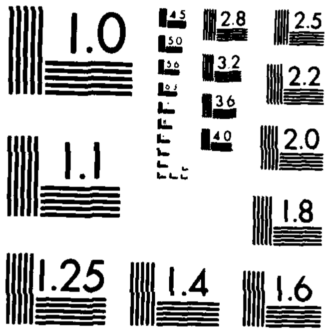
AD-A151 718   AUTOMATIC VLSI (VERY LARGE SCALE INTEGRATION) ROUTING   1/2
USING 2-LAYER METAL(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..   T G HEWITT
UNCLASSIFIED   DEC 83 AFIT/GCS/EE/84S-2   F/G 20/12   NL






MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

# AUTOMATIC VLSI ROUTING USING 2-LAYER METAL

THESIS

AFIT/GCS/EE/84S-2 Terry G. Hewitt
Capt USAF





DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/EE/84S-2





# AUTOMATIC VLSI ROUTING USING 2-LAYER METAL

THESIS

AFIT/GCS/EE/84S-2 Terry G. Hewitt

Capt USAF

AFIT/GCS/EE/ 84S-2

# AUTOMATIC VLSI ROUTING
# USING 2-LAYER METAL

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Terry G. Hewitt, B.S.

Capt USAF

Graduate Computer Systems

December 1983

## Acknowledgments

I would like to thank my advisor LTC Harold Carter and my reader LTC Gerald Armstrong for all of the help that I received from them. I would like to especially thank my wife for all of the support she gave me throughout the entire thesis effort.

Contents

Contents

## List of Figures

## Abstract

Automatic routing of a computer chip is a complex task. When routing and VLSI (Very Large Scale Integration) design are combined, the problem is increased.

A computer program was developed to automatically route the interconnections of a VLSI chip. Only two point nets can be routed using the program. The interconnections were routed using a "dogleg" channel router on both horizontal and vertical channels. The program runs very quickly. Fifty nets were routed in less than 1 second.

The program minimizes the channel height of a channel. The channels must be rectangular. Also, each horizontal channel must intersect every vertical channel and vice versa.

Alternate paths can be found for nets in horizontal channels when channel capacity is exceeded. Constraint loops are removed by ordering the way nets are routed or by introducing a "dogleg".

The program produces output that is compatible with CLL (Chip Layout Language). The output from the program can be merged with CLL statements that place cells from a library on a grid to form plots or to create CIF (Caltech Intermediate Form) data to be used in making VLSI chips.

# AUTOMATIC VLSI ROUTING USING 2-LAYER METAL

## I. Introduction

Automatic routing of a computer chip is a complex task.. When routing and VLSI (Very Large Scale Integration) design are combined, the problem is increased.

A VLSI designed chip will have more interconnections because it will have more cells. The more interconnections, the longer it will take to route the chip. The routing algorithm chosen must minimize the time required to accomplish this task.

A VLSI chip also has multiple routing layers which increases the complexity of the problem.

### Routing

Before automatic routing can be discussed it is necessary to understand fully how routing is accomplished. There are four steps involved in routing (Akers, 1972:287). The four steps are:

1. wire-list determination
2. layering
3. ordering
4. wire layout

Wire-list determination. The first step is to identify what modules and pins are to be connected. An example would be connect Module 1 pin 3 to Module 4 pin 7. A connection or net may specify more than two locations. Normally this information will be given for the routing problem.

Layering. When multiple layers exist, the chip designer decides on which layer a wire begins and ends. However, the path between these points is up to the routing algorithm. By making use of multiple layers it is possible to reduce congestion on any one layer. Reducing the congestion makes modifications to the final design easier and increases potential packing of devices on the chip. Congestion can be reduced by dividing the wires evenly between the layers. Another approach is to break up a net into horizontal and vertical wire segments. Each type of wire segment is then dedicated to a separate layer.

Ordering. After all the wires have been assigned to a layer, it is necessary to determine in what order the wires will be laid out. Different ordering schemes exist that range from being very specific to having no order at all (Hightower, 1973:4). The right ordering scheme can reduce the average capacitance per net (Deutsch, 1976:427). The scheme used is determined by which routing algorithm is chosen.

Wire Layout. In this step the routing algorithm is applied. It is necessary for the algorithm to define paths

for wires in an efficient manner. The most important criteria is that as many wires are automatically routed as possible. Other criteria to judge efficiency would be how much time and computer memory are required to complete routing.

VLSI Design

VLSI electronic circuitry may contain hundreds of thousands of transistors on a single silicon chip. These chips represent integrated systems more than integr circuits. Integrated systems that use MOS (metal-oxide-semiconductor) technology contain multiple layers. These layers are termed metal, polysilicon, and diffusion. Pathways on the different layers and the location of vias (connections between the layers) are transferred to the layers during the fabrication process from masks similar to photographic negatives.

Paths on the metal layer may cross over paths on either the polysilicon layer or the diffusion layer with no significant functional effect. However, when polysilicon crosses diffusion, a transistor is created. The transistor has the same characteristics as a simple switch (Mead, 1980:1).

To identify an end point of a path the x and y coordinates are specified as is the layer of the end point. To identify an interconnection both end points must be

specified.

## Problem and Scope

The problem is to automatically route the interconnections on a VLSI chip. To accomplish this, a new program has been written that can be used in conjunction with CLL (Chip Layout Language).

CLL is a manual method for describing VLSI design. With the addition of the new program the chip designer is able to automatically route the interconnections on a VLSI chip.

The routing program developed in this study solves only a part of the total routing problem. Only two point nets are automatically routed.

The analysis and design of this study are limited to three items:

1. Choosing a routing algorithm that minimizes wire length.

2. Developing a new program for use with CLL to assist the routing process.

3. Integrating the two programs so that partial automatic routing is accomplished.

## Assumptions

It is assumed that the user of this routing tool is familiar with CLL and VLSI design. Also that the chip specified can be routed. The user should realize that there is an optimum placement for modules such that the total

The simplest case is when the endpoints share the same channel. One segment connects the two endpoints. When the endpoints do not share the same channel it is slightly more complicated. Starting at one endpoint follow the horizontal channel until the closest vertical channel is found. If the endpoints still can not be connected follow this channel until the closest horizontal channel is reached and connect the paths.

Assign Tracks. In Assign Channels each net is logically placed in the center of a channel. Assign Tracks assigns each segment of a net to a free track in a channel. There are three steps in the algorithm which assigns a segment to a track: 1) resolve conflicts, 2) check channel capacity, and 3) route the channel.

Resolve Conflicts. The first step is to resolve any conflicts between segments. Resolving conflicts eliminates constraint loops that can cause a net to be unroutable. Constraint loops can be avoided by assigning a priority to segments or by introducing a dogleg. There are three types of constraint loops.

Routing

Routing is broken up into three parts, Assign Channels, Assign Tracks, and Form CLL Statements. Assign Channels finds a path between endpoints named by the CONNECT statement. Assign Tracks finds a specific track on a channel for an interconnection. Form CLL Statements creates a file of CLL statements from the wire segments that have been assigned a track.

Assign Channels. In this part of the program a path is found for each net. This path is made up of horizontal and vertical wire segments. Each segment is recorded in the proper channel by its endpoints. The track capacity of the channel is ignored at this time.

To find a routing path the first step is to find out which channels the endpoints are in. A segment is then extended from the endpoint to the center of the channel. It may take several steps to go from the endpoints to the center of their respective channels. The endpoints can be on any layer. The segment from the endpoint to the center of the channel can not be on the routing layer. The segment can not be on the metal layer for horizontal channels or the metal2 layer for vertical channels. It may be necessary to change layers and generate a CLL VIA statement.

The second step finds a path that connects the two endpoints together. The algorithm follows.

within horizontal channels are routed before those in vertical channels. The order in which the horizontal channels are described determine the order that the wires within a channel are routed. That is, the wires in the first horizontal channel described are routed before the wires in the second horizontal channel described are.

The format for horizontal channel input follows.

```
BEGIN-HCHANNELS
corner point    opposite corner point
/* all horizontal channels described here */
END-CHANNELS
```

The format for vertical channel input follows.

```
BEGIN-VCHANNELS
corner point    opposite corner point
/* all vertical channels described here */
END-CHANNELS
```

Corner point and opposite corner point are the x and y coordinates that describe a channel.

Net Input. The last type of input is a description of what is to be connected. This description must include the x and y coordinates as well as the layer of the endpoints. The format of the CONNECT statement is:

```
CONNECT x y layer x y layer
```

Layers can be any combination of the following:

diff (diffusion)
poly (polysilicon)
poly2
metal
metal2

If the routing layers are not specified the program will route on any or all of metal, metal2, poly, and diff. Metal and metal2 layers must always be included.

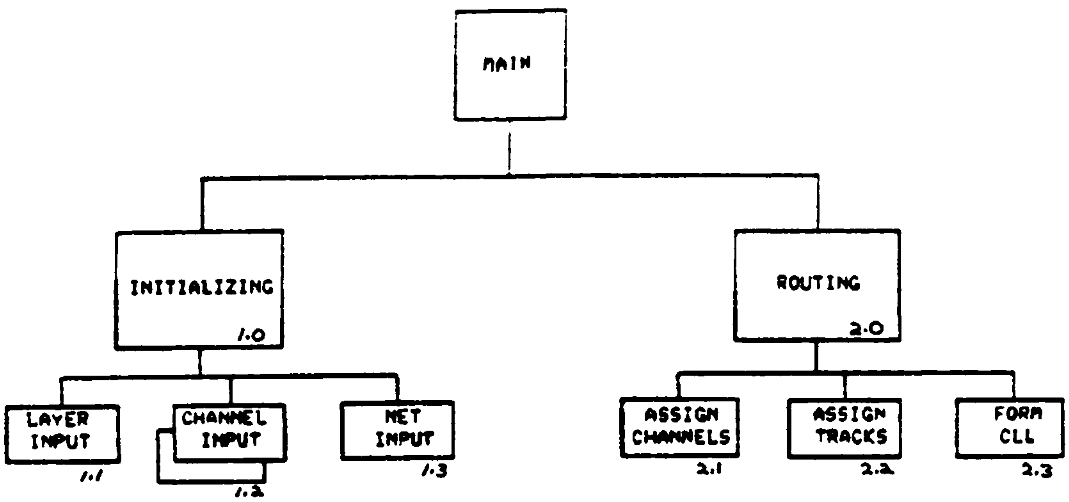



Fig. III-1. The Automatic Routing Program

<u>Channel Input</u>. It is a requirement that all channels be rectangular, therefore only two points are needed to describe a channel. These two points must be on opposite corners, either the top left and bottom right or bottom left and top right.

There are two types of channels, horizontal and vertical. Horizontal channels are channels that have endpoints of nets on the north and south side. Vertical channels are channels that have endpoints on the east and west side. The wires

# III. System Design

This chapter states how the requirements defined previously are met to automatically route the interconnections of a VLSI circuit.

The routing program is broken up into two areas: initializing and routing (See Figure III-1). Initializing accepts three types of input: layer input, channel input, and net input. Routing routes the nets along the channels and creates CLL WIRE and VIA statements that describe the routing path. This output file is manually merged with another CLL file that contains statements for cell placement. The two files together complete the circuit design.

## Initializing

This part of the program analyzes the input file for the rest of the program. All three types of input are contained in a single input file.

Layer Input. The first step is to define the routing layers that make up the VLSI circuit. The format to specify the routing layers follows.

```
BEGIN-LAYERS
layers
END-LAYERS
```

avoided.

This routing program is only a partial implementation of a total circuit design program. Only two point nets are routed. Nets that contain more than two endpoints must be done manually or broken into two point nets.

These CLL statements describe the routing path of the different nets. Each net's CLL statements will be preceded by a comment that will give the source point and destination point of that net. The output file is then merged with the CLL program and the circuit design completed .

This output file is modifiable and can be updated if needed. Small design changes can be made without repeating the entire routing process.

If an error is detected while processing routing information, the program is halted and an appropriate error message is printed. Errors can occur from illegal input or from violation of design constraints. Design constraints can be avoided by making sure that the routing channels are big enough.

Routing Algorithm. To find a path between endpoints a routing algorithm is used. Each net is assigned to channels by the routing algorithm. The wire segments within the channels are then routed using the "dogleg" channel router. This algorithm assumes that the endpoints share the same channel.

## Additional Requirements

The routing path of the nets will follow the design rules prescribed by Mead and Conway (Mead, 1980:47-51). These rules prescribe the minimum distances between wires and layers. By following these rules many layout errors can be

is a requirement because of the search algorithm used to connect endpoints.

Layer Description. The path between endpoints is made up of horizontal and vertical segments. The horizontal segments lie on the metal layer. The vertical segments lie on the metal2 layer. By using the metal layers capacitance is reduced and unwanted transistors are avoided. The endpoints, however, can lie on any layer. The routing program must know what routing layers are available so that the endpoints can be connected to the routing path. For the automatic routing to be accomplished, the metal and metal2 layers are required.

Net Description. After the routing layers and channels have been described, the nets are described. A net is made up of two endpoints that must be connected. Each endpoint is described by its x and y grid location and layer. The endpoint must lie on or outside the channel boundary. If the endpoint lies outside the channel it must be closer to its channel than any other. The layer of an endpoint must have been previously described in the layer description.

The Output. The output of the routing program will be a file that contains only CLL comment, WIRE and VIA statements. The WIRE statement is how CLL connects two points that lie on the same routing layer. The VIA statement connects two points that have the same x and y coordinates but lie on different routing layers.

# II. Requirements

The main requirement is to automatically route interconnections between active regions on a VLSI integrated circuit. With CLL it is possible to design and route an integrated circuit manually. In order to route the interconnections automatically, a new program must be created that will work in conjunction with CLL. Of the routing methods available, channel routing seems to be the most applicable. The routing program developed by this thesis uses the dogleg channel router.

The CLL program is written in C and is implemented on the SSC VAX 11/780. The routing program developed as a part of this thesis interfaces with CLL. Therefore, it is preferable that it too use the same language and computer.

## Automatic Routing Program

The routing program's function is to find a path between the endpoints of a net. To find a path three pieces of information are needed: channel description, layer description, and net description. Once a path has been found it is transformed into CLL statements that can be used by the CLL program.

Channel Description. Channels are rectangular and are described by their corner points. Each horizontal channel must intersect every vertical channel and vice versa. This

The router runs vertical and horizontal expansion lines from the two terminals to be connected. Then for each line, it finds the longest perpendicular escape line. This process is repeated until expansion lines, one from each terminal cross. In most cases, the algorithm generates the path with the minimum number of bends (fewer vias) (Soukup, 1981:1295).

The Hightower algorithm is fast for simple mazes. However for complicated mazes it is slow, needs a large stack of data, and does not guarantee a connection if it exists (Soukup, 1981:1295).

Conclusions. The "dogleg" channel router will be used for this project. The algorithm is fast and easy to apply. Also, channel routing algorithms work best when there are multiple layers. This routing tool is to be used by students at AFIT so speed and quick turn around are more important than optimal design. The other algorithms would require too much time to be useful and are more difficult to implement.

## Approach and Presentation

The requirements will be presented in Chapter II. In Chapter III the system design is laid out. In Chapter IV a complete detailed design is given and in Chapter V the conclusions and recommendations are presented.

The algorithm follows. To limit the number of doglegs, only allow doglegs at terminal positions. Doglegs would not be allowed on a two terminal net unless it is needed to resolve a constraint loop. A three terminal net could be doglegged only once, and so on. Next, order the terminals within a net based on their abscissas and decompose the original net into a series of two terminal subnets such that the nth subnet consists of terminals n and n+1. When a subnet ends the next subnet of the same net can be placed on the same track. To minimize the amount of doglegs a "range" can be added. This range represents the minimum number of consecutive subnets that must be assigned to a track. As the range gets larger fewer doglegs are allowed (Deutsch, 1976:427).

The channel router has a limitation that must be accounted for (Soukup, 1981:1295). Terminals in the channel may create a constraint loop. An example of a contraint loop would be two nets blocking a third net from being routed.

Linear Expansion. The Hightower algorithm is different from Lee's in that the whole grid is not stored in a matrix. Instead only lines and points are stored. The algorithm connects a pair of points by constructing a sequence of line segments emanating from each point. When two segments intersect, a path has been found. Then a retrace algorithm finds the shortest path back to the starting points (Hightower, 1973:8-9).

The big disadvantage of using Lee's algorithm is the amount of time needed to complete the routing. The algorithm does guarantee a minimum path if one exists and the speed improves as the area gets more congested (Soukup, 1981:1295). Because of the great number of interconnections on a VLSI chip the time to route is prohibitive using this algorithm.

Channel Router. A channel router is different from Lee's router in that the channel router operates on one routing track at a time so that a smaller amount of data must be core resident. A channel is defined as the space between two terminals. A terminal can be a pin on a PLA or a Cell from the Cell Library. A net consists of two or more terminals that must be connected via some routing path. The routing will take place on two levels, horizontal segments on one level and vertical segments on another. The channel length is the distance between the terminals and this can not be changed. It remains for the router to minimize channel height; that is, the spacing between the horizontal tracks.

A variation of the channel router is the "dogleg" channel router (Deutsch, 1976:425). This is a channel router with a difference. The "dogleg" router allows for more than one horizontal segment per net. However, doglegs increase the apparent local density and the corresponding added contacts increase the capacitance (generally an undisirable result) so the number of doglegs should be kept at a minimum.

obstructions exist the problem becomes more difficult.

The first step is to [illegible] the two points. Then enter a 1 in each empty cell adjacent to the starting point. Next enter a 2 in each empty cell adjacent to the 1's. Next, 3's are entered adjacent [illegible] and so on. Continue this process until the destination [illegible] reached. When the destination is reached a path [illegible] along with its length. All that remains is to [illegible] the source by finding the cell with the [illegible]. When two equal paths exist then it is arbitrary which [illegible] chosen (Akers, 1972:312-314).

There are some improvements that can be made to make this algorithm even more efficient. The direction should not be changed unless its necessary. A priority scheme is used to change direction. Something simple such as N-E-W-S north first, east second, west third, and south last. This will lead to a uniform nesting effect (Akers, 1972:314).

When choosing the starting position choose the one furthest from the center of the board. This will cause less of the grid to be searched. Another method would be to start the search from both points. This would have a little more overhead but the path would be found quicker assuming the search could start from both points simultaneously. Also an artificial bound around the two points would restrict the grid so that a smaller area would have to be searched (Akers, 1972:317).

interconnection distance is minimized (Breuer, 1972:18). The routing algorithm may work better for some placements than others.

Error checking will be kept at a minimum. The new program will check for correct syntax and flag any nets that can not be routed due to design constraints. Additional errors will be caught by using two other programs, DRC (Design Rule Checker) and ESIM (Switch-level simulator).

## Literature Review

It is important to find a routing algorithm that will be efficient and fast. This study is automating the process to gain speed. There are three main algorithms used to route interconnections (Soukup, 1981:1295).

1. Grid Expansion
2. Channel Router
3. Linear Expansion

Grid Expansion. Lee's algorithm is a technique that was derived from the shortest-path algorithm used in operations research and graph theory. The algorithm is based on expanding a wave from one point to another. At each step, grids on a diamond-shaped front wave are expanded one step further (Lee, 1961:346-365).

The problem is to find the shortest distance between two points. If there were not any obstructions, then it would be easy. A straight line can easily be found. But when

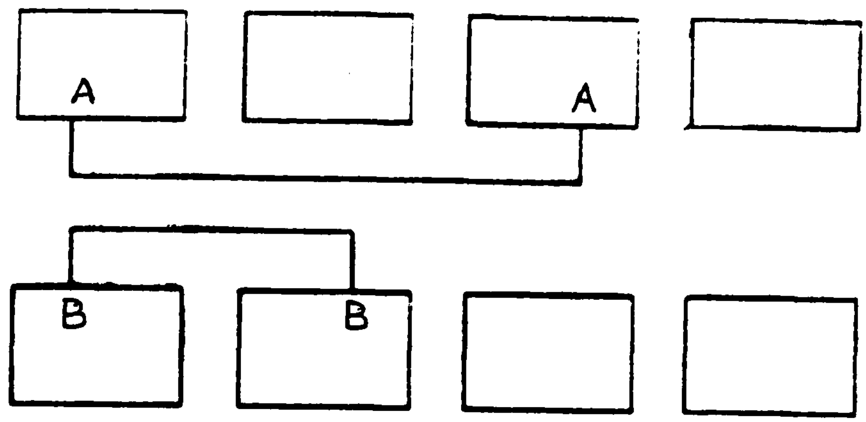



Fig. III-2. Type 1 conflict

Type 1 (See Figure III-2) occurs when two different nets begin at the same point on opposite sides of a channel. When this occurs the net on top must be assigned a track above the lower net. This is done by routing that segment first.

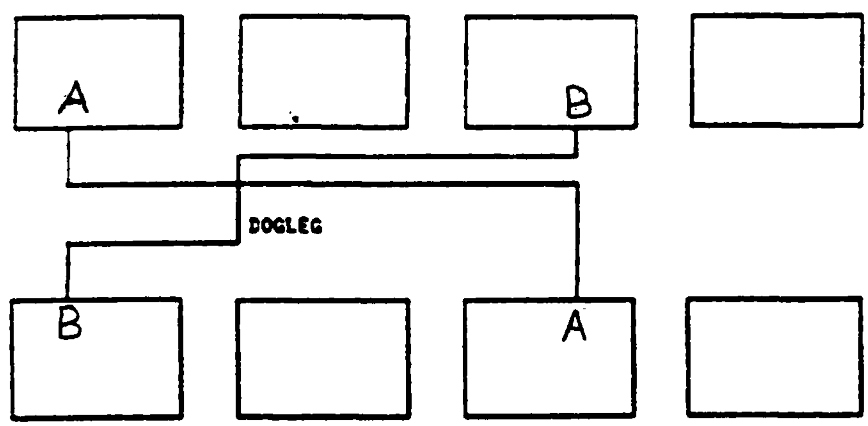



Fig. III-3. Type 2 conflict

Type 2 (See Figure III-3) occurs when two nets start on opposite sides of the channel and also end at the same point

only on opposite sides of a channel. When this occurs one of the nets must be doglegged.

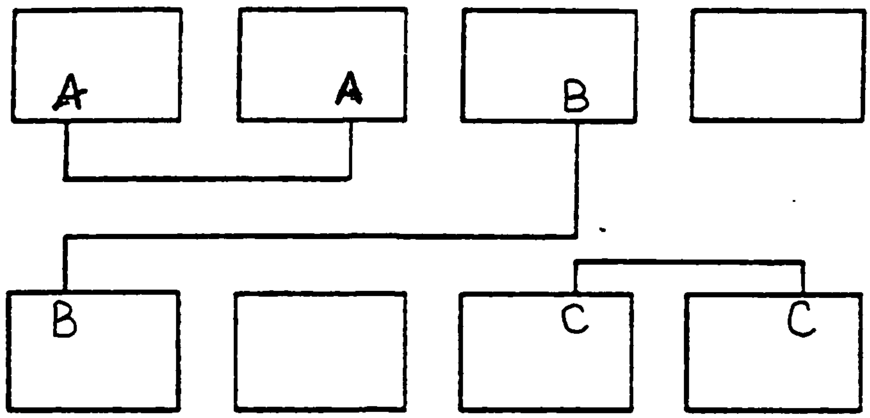



Fig. III-4. Type 3 conflict

Type 3 (See Figure III-4) occurs when nets start and end at the same point. In this case several things must happen. Net A must be above Net B and Net B must be above Net C. Net A is assigned a higher priority than Net B which is assigned a higher priority than Net C.

Check Channel Capacity. The second step is to check and make sure that the channel capacity is not exceeded. If the channel capacity is exceeded then an alternate routing path must be found for some of the nets in that channel. To check channel capacity the following algorithm is used.

There is a mathematical lower limit to the number of tracks needed to route a channel (Deutsch, 1976:426). To find this limit, the limit of each net is found first. A net's limit is found by finding the leftmost and rightmost x

coordinate of a net for horizontal channels (a similar description holds for vertical channels as well). Then for each terminal position in the channel, the number of nets whose extent includes that terminal's x coordinate is counted. The maximum net limit found in a channel is the number of tracks needed to route that channel. If the available tracks are equal to or below this limit routing continues. If available tracks exceed the limit a new path must be found for some of the nets in this channel.

Route Channel. The nextstep is to implement the "dogleg" channel router. The wires within a channel are routed a channel at a time. Each wire segment in a channel is assigned a specific track within the channel. If any net can not be routed then the program is halted and an error message is printed that identifies the channel that was to small.

Form CLL Statements. The final part of the routing program is where each net is transformed into CLL WIRE and VIA statements. The CONNECT statement of each net preceeds the WIRE and VIA statements as a comment. This makes it easier to modify the output file if necessary.

# IV. Detail Design

This chapter discusses the automatic routing program in detail. The first part of this chapter outlines the special C structures that are used. The last part outlines each module of the routing program.

## Special Structures

The routing program is written using the C language. In C a structure is a collection of one or more variables. In this program the structure construct is used to hold channel and segment information.

The channel structure holds all of the information that pertains to a channel.

```
struct channel (
                int     id;
                int     done;
                int     center;
                int     luseg;
                int     ltseg;
        struct point   corner;
        struct point   opcorner;
        struct wireseg  track();
        struct wireseg  untrack();
);
```

The variable id identifies the channel to be horizontal if equal to 0, or vertical if equal to 1. The variable done is a boolean flag that is set to TRUE when the channel is ready for final routing. The variable center contains the location of the center of the channel. The variable luseg is the upper limit of the untrack array. The variable ltseg is the upper

limit of the _track_ array. _Corner_ and _opcorner_ contain the x and y coordinates of the corner points of a channel. The point structure holds an x and y coordinate.

```
struct point (
     int  xloc;  /* x coordinate */
     int  yloc;  /* y coordinate */
);
```

_Track_ and _untrack_ are arrays of the wireseg structure. The wireseg structure holds all of the information for a wire segment. The _track_ array contains the wire segments that are routed. The _untrack_ array contains the segments that are not routed.

```
struct wireseg (
     int  tag:
     point  leftend;
     point  rightend;
     wireseg  *left;
     wireseg  *right;
);
```

The integer variable _tag_ is a boolean that is set to 1 when that segment is routed. _Leftend_ and _rightend_ hold the x and y coordinates of the segment endpoints. _*left_ and _*right_ point to the previous and next segments of a net.

There is one additional structure that holds all the information for a net.

```
struct net (
            char   layer(2);
     struct point  start;
     struct point  end;
     struct *wpoint;
     struct channel  *pointer(2);
);
```

The character array _layer_ holds a letter signifying what layers the endpoints are on: m for metal, p for poly, d for

diff, 2 for metal2, and P for poly2. _Start_ and _end_ hold the x and y coordinates of the endpoints. _*wpoint_ points to the first wire segment for that structure. _*pointer_ points to the channel that the endpoints are in.

## _The Routing Program_

This program is developed using a software engineering technique called top down design. The upper modules are high level and control the order in which other modules are called. The lower modules are low level and accomplish a specific task. By using this technique the routing program can be implemented one module at a time. This increases software reliability by making testing and debugging easier.

_Auto Route VLSI_. This is the main routine. Its function is to call the initializing routines and then call the routing routines.

Input: None

Output: None

Functions called: initializing routing

Calling functions: None

Notes: See Figure III-1. This routine also initializes global variables that can not be initialized when the variables are declared.

Psuedo code:

```
read the input
route the nets
form CLL statements
```

Initializing(1.0) This routine processes the input file. It decides which of three types of input is being processed and calls the appropriate subroutines.

Input: An input file that contains three types of input: layer, channel, and net.

Output: None

Functions called: layer_input, channel_input, and net_input

Calling fuctions: Auto_Route_VLSI

Notes: See Figure IV-1

Psuedo code:

```
while there is input
  get keyword
  if LAYER then
    process later input
  else if CHANNEL then
    process channel input
  else if NET then
    process net input
  else
    ERROR
```

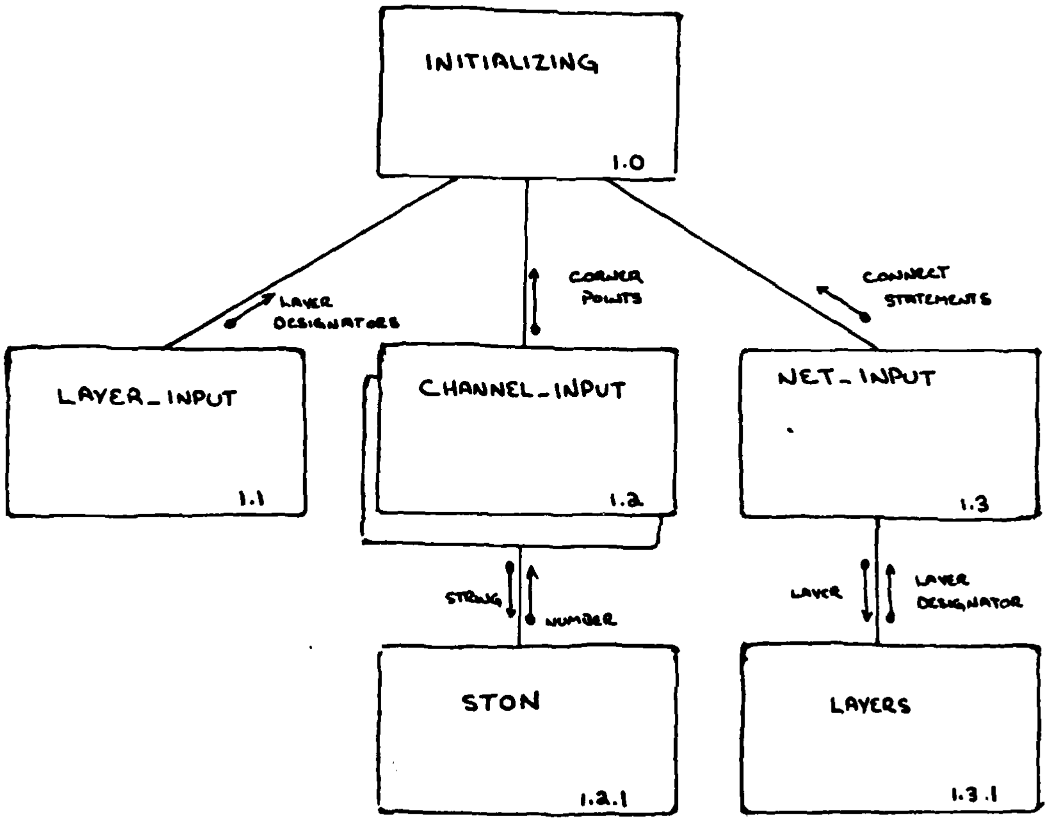



FIG. IV-1. THE INITIALIZING ROUTINES

__Layer Input__(1.1) This routine accepts routing layer information. Two routing layers, metal and metal2, are required. A boolean flag is set to 1 for each routing layer specified.

Input: A statement of the form "BEGIN-LAYER layers END-LAYER".

Output: Boolean variables are set to 1 for specified routing layers.

Functions called: None

Calling functions: initializing

Notes: The default is four layers: metal, metal2, poly, and diff. If layer input is different than the default, it must precede channel and net input.

Psuedo code:

```
turn off default layers
get keyword
while processing layer input
  if METAL then
    turn metal layer on
  else if METAL2 then
    turn metal2 later on
  else if DIFFUSION then
    turn diffusion layer on
  else if POLY then
    turn poly later on
  else if POLY2 then
    turn poly2 later on
  else
    ERROR
  get keyword
if both metal layers not on then
  ERROR
```

__Channel Input__(1.2) There are two routines that accept channel description information, one routine for horizontal channels and one for vertical channels. Each type of channel

is held in an array of channels. The corner points are recorded and the center of the channel is calculated and recorded.

Input: A statement of the form:

"BEGIN-HCHANNELS

corner point    opposite corner point

END-HCHANNELS"

or

"BEGIN-VCHANNELS

corner point    opposite corner point

ENDVCHANNELS"

Output: Each channel is held in an array of horizontal channels or vertical channels.

Functions called: ston

Calling functions: initializing

Notes: A program, ston(), is used to convert corner points from character strings to integers.

Psuedo code:

```
while processing channel input
  store grid location of corner
  store grid location of opposite corner
for all channels
  find midpoint of channel
```

Net Input(1.3) This routine accepts CONNECT statements that describe the nets to be routed. The nets are held in an array of nets. The endpoints of the nets are recorded and the routing layer designator for each point is calculated.

Input: A statement of the form:

"CONNECT x y layer x y layer"

Output: Each net is placed in an array of nets.

Functions called: layers

Calling functions: initializing

Notes: A program, layers(), returns a single character designator defining the endpoint layer. It also checks to see if the layer is available.

Psuedo code:

```
store grid location of start of net
store layer designator of start of net
store grid location of end of net
store layer designator of end of net
```

Routing(2.0) After the input has been processed the routing routines are called. Assign channels finds a path between the source and destination of a net. Assign tracks finds a specific track for the path. Form CLL creates a file for output of CLL WIRE and VIA statements.

Input: None

Output: None

Functions called: assign_channels, assign_tracks, and form_CLL

Calling functions: Auto_Route_VLSI

Notes: See Figure III-1

Psuedo code:

```
find a path for a net
assign wire segments specific track locations
form CLL statements
```

Assign Channels(2.1) The purpose of this routine is to route a path between endpoints of a net. A path is found from each endpoint to the center of its starting or ending channel. A path is then found to complete the net. The path follows channels and does not concern itself with track capacity.

Input: A description of the nets found in the net array.

Output: A routing path is stored in the channel structure.

Functions called: find_channel, center_channel, and find_channel_path

Calling functions: routing

Notes: See Figure IV-2

Psuedo code:

```
find channel of start and end of net
find path to center of channels
find path to complete nets
```

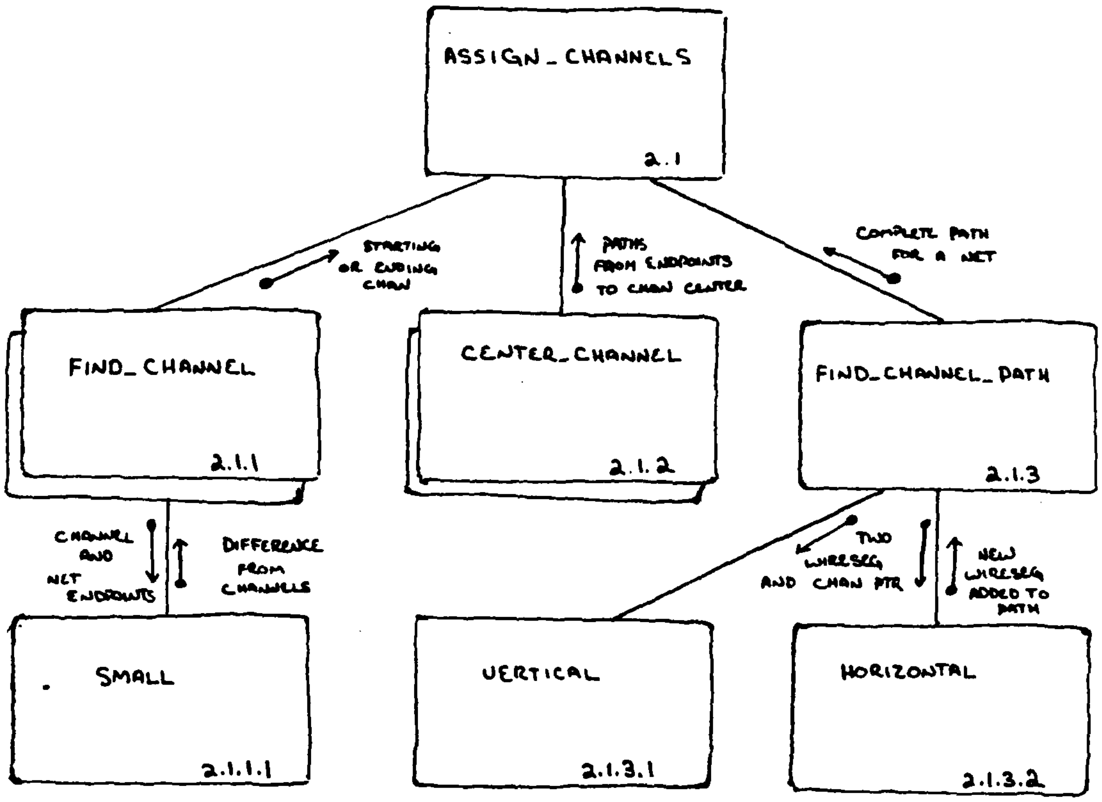



FIG. IV-2. THE ASSIGN CHANNEL ROUTINES

*Find Channel*(2.1.1) There are two routines that find the channel. One routine finds the channel of the start point of a net, the other finds the channel of the end point of the net. The horizontal channels are checked first. If the difference from the endpoint is 0, the channel is found. If not 0, the closest channel is kept and the vertical channels are checked.

Input: A description of the nets found in the net array.

Output: The variable *pointer* contains the address of the channel.

Functions called: small

Calling functions: assign_channels

Notes: A routine, small(), determines how far the net endpoint is from the channel.

Psuedo code:

```
for all nets
  for all horizontal channels
    find difference from channel boundary
    if difference = 0 then
      stop channel found
    else
      keep smallest difference
  for all vertical channels
    find difference from channel boundary
    if difference = 0 then
      stop channel found
    else
      keep smallest difference
```

*Center Channel*(2.1.2) There are two routines that route a segment from the endpoints to the center of its channel. One routine routes a segment for the start point of a net and another routes a segment for the end point of a net. These

Router(2.2.4.1) There are two routines that check for Type 3 conflicts and route wire segments, one for horizontal wire segments and one for vertical segments.

Type 3 conflicts are resolved by checking the segment to be routed against all other unrouted segments. If the leftend of the segment to be routed is equal to the rightend of some other unrouted segment, or if the rightend of the segment to be routed is equal to the leftend or rightend of an unrouted segment, a Type 3 conflict can occur. If the unrouted segment connects with the top of the channel, the segment to be routed is skipped. If a horizontal segment is being routed and the endpoint lies within a vertical channel a Type 3 conflict can not occur and the segment can be routed.

If the segment to be routed does not have a Type 3 conflict the segment is assigned a unique track location and that segment is marked routed.

Input: The two inputs to this routine are pointers to the wire segment and the address of the channel it belongs to.

Output: A wire segment is assigned a unique track location.

Functions called: go_up and chknpt

Calling functions: route

Notes: The program chknpt checks to see if the endpoint lies within a vertical channel. The routine is called only when routing horizontal segments.

Psuedo code:

```
for all channels
  if channel has not been routed yet
    find difference between channels
     not already routed
    keep closest channel
if no channel found then
  ERROR
build new segment in new channel
adjust pointers
```

Route(2.2.4) There are two routines that route the wire segments in the channels. One routine routes horizontal channels and the other routes vertical channels. Each routine finds a unique location for each wire segment after checking for Type 3 conflicts. Horizontal channels are routed before vertical channels. Tracks are assigned from the top for horizontal channels and from the right for vertical channels.

Input: A channel index

Output: The segments in a channel are assigned to a track and all pointers are adjusted.

Functions called: router

Calling functions: assign-tracks

Notes: None

Psuedo code:

```
find top of channel
for all wire segments
  get first segment
  if segment will fit then
    check for Type 3 conflict
    route
  if still segments to route then
    increment channel top pointer
    adjust pointers
```

routed already is used. All x and y coordinates are adjusted for that net. The segment replaced is removed.

Input: A channel index

Output: A segment is added to another channel to form a new path for a net.

Functions called: new_segment

Calling functions: check_capacity

Notes: None

Psuedo code:

```
for all segments in a channel
  find a segment with both endpoints in
                     a vertical channel
  create new segment to take its place
if no new segment created
  ERROR
```

<u>New Segment</u>(2.2.3.2.1) The <u>new segment</u> routine builds a new segment for an alternate path. The routine <u>alternate paths</u> finds a segment to be replaced. The <u>new segment</u> routine finds a location for the new segment. After the new segment has been found all of the pointers are adjusted and the old segment is deleted.

Input: There are three inputs to this routine. They include a pointer to the channel, the wire segment to be removed, and an index to the sorted array for that segment.

Output: A new wire segment is added and an old one is deleted.

Functions called: None

Calling functions: alternate_paths

Notes: None

for all segments and the maximum found is the tracks needed.

Input: A channel index

Output: The maximum tracks needed

Functions called: None

Calling functions: check_capacity

Notes: None

Psuedo code:

```
for all wire segments
  for each wire segment
    count the number of segments that
                                include it
    return high number of tracks
```

<u>Alternate Paths</u>(2.2.3.2) This routine is called when the tracks needed exceeds the track capacity of a horizontal channel. An alternate path can not be found for vertical channels because the horizontal channels have already been routed. By finding an alternate path tracks needed are reduced. There is one requirement to remove a segment. The segment removed must be a middle segment of a net. That is, it can not contain an endpoint of a net. It is preferable that the segment removed is a long one. A longer segment would potentially reduce track capacity quicker. After each segment is removed the tracks needed to route are calculated again.

After a segment has been selected for removal the following algorithm is used to find a new path for that net. A segment from a horizontal channel is moved to another horizontal channel. The closest channel that has not been

Psuedo code:

```
while between endpoints
  increment a distance
  if new location does not cause conflict
    stop
```

<u>Check Capacity</u>(2.2.3) There are two routines that check channel capacity. One routine is for horizontal channels and one for vertical channels. These routines count how many tracks are available. A routine is called that counts the tracks needed to route. If tracks needed exceed tracks available, a routine is called to reduce tracks needed. When a channel has been checked for track capacity successfully the boolean flag <u>done</u> is set to TRUE for that channel.

Input: A channel index.

Output: The tracks needed is compared with tracks available.

Functions called: tracks_needed and alternate_paths

Calling functions: assign_tracks

Notes: None

Psuedo code:

```
find top and bottom of channel
for all wire segments
  if segment had to change layers
    reduce top or bottom accordingly
count tracks available
count tracks needed
while tracks available less than tracks needed
  reduce tracks
  count tracks needed
```

<u>Tracks Needed</u>(2.2.3.1) There are two routines that count the tracks needed, one for horizontal channels and one for vertical channels. The count is calculated by counting the segments that include an endpoint of a segment. This is done

by starting location. Rather than sort the wire segment array, and array of pointers to the wire segment array is sorted.

Input: The inputs are channel index and a starting position.

Output: Sorted array.

Functions called: None

Calling functions: resolve_conflicts

Notes: None

Psuedo code:

```
if sorting starts at beginning
  initialize sort array
else
  initialize sort at new beginning
while sorting is not done
  for all segments to sort
    compare sets of segments
    switch and swap
```

New X and New Y(2.2.2.2) These two routines find a location for a dogleg that does not cause a new conflict. New X finds a location for horizontal channels and New Y finds a location for vertical channels.

Input: The three inputs are a channel pointer, wire segment pointer, and a pointer into the sorted array.

Output: The output is a location for a dogleg.

Functions called: None

Calling functions: resolve_conflicts

Notes: None

The segments are checked to see if they have the same starting position. When this condition is true the ending points are checked to see if they connect to opposite sides of the channel. If they do, it identifies a Type 2 conflict and if they do not it identifies a Type 1 conflict.

Type 1 conflicts are resolved by making sure the upper net is routed before the lower net. This can be done by having the upper net first in sorted order.

Type 2 conflicts are resolved by dividing the lower segment into two segments. This introduces a dogleg which resolves the constraint.

Input: A channel index.

Output: The segments in a channel are resorted to resolve conflicts.

Functions called: sort_loc, new_x, and new_y

Calling functions: assign_tracks

Notes: None

Psuedo code:

```
sort segments within channel by starting loc
for all wire segments within a channel
  if two wire segments start in same loc
    let wire segment connecting upward
    be first in sorted order
  else
    if two wire segments end in the same loc
      create a dogleg
  sort segments within channel by starting loc
```

<u>Sort Loc</u>(2.2.2.1) There are two routines that sort the wire segments within a channel, one for horizontal channels and one for vertical channels. The wire segments are sorted

Psuedo code:

```
sort wire segments by starting locations
for all horizontal channels
  resolve Type 1 and Type 2 conflicts
  check channel capacity
  route wire segments
for all vertical channels
  resolve Type 1 and Type 2 conflicts
  check channel capacity
  route wire segments
```

<u>Sort Points</u>(2.2.1) This routine sorts the points within a wire segment. Horizontal wire segments are sorted by x location and vertical wire segments are sorted by y location. The smallest value is stored in <u>leftend</u> of the channel structure.

Input: The wire segments within the channel.

Output: Sorted wire segments within a channel.

Functions called: None

Calling functions: assign_tracks

Notes: None

Psuedo code:

```
for all horizontal channels
  for all wire segments within a channel
    sort wire segments by location
for all vertical channels
  for all wire segments within a channel
    sort within wire segments by location
```

<u>Resolve Conflicts</u>(2.2.2) There are two routines that resolve Type 1 and Type 2 conflicts. One routine resolves convlicts for horizontal channels and one resolves conflicts for vertical channels. These routines isolate what type constraints a channel has, if any, and resolves it.

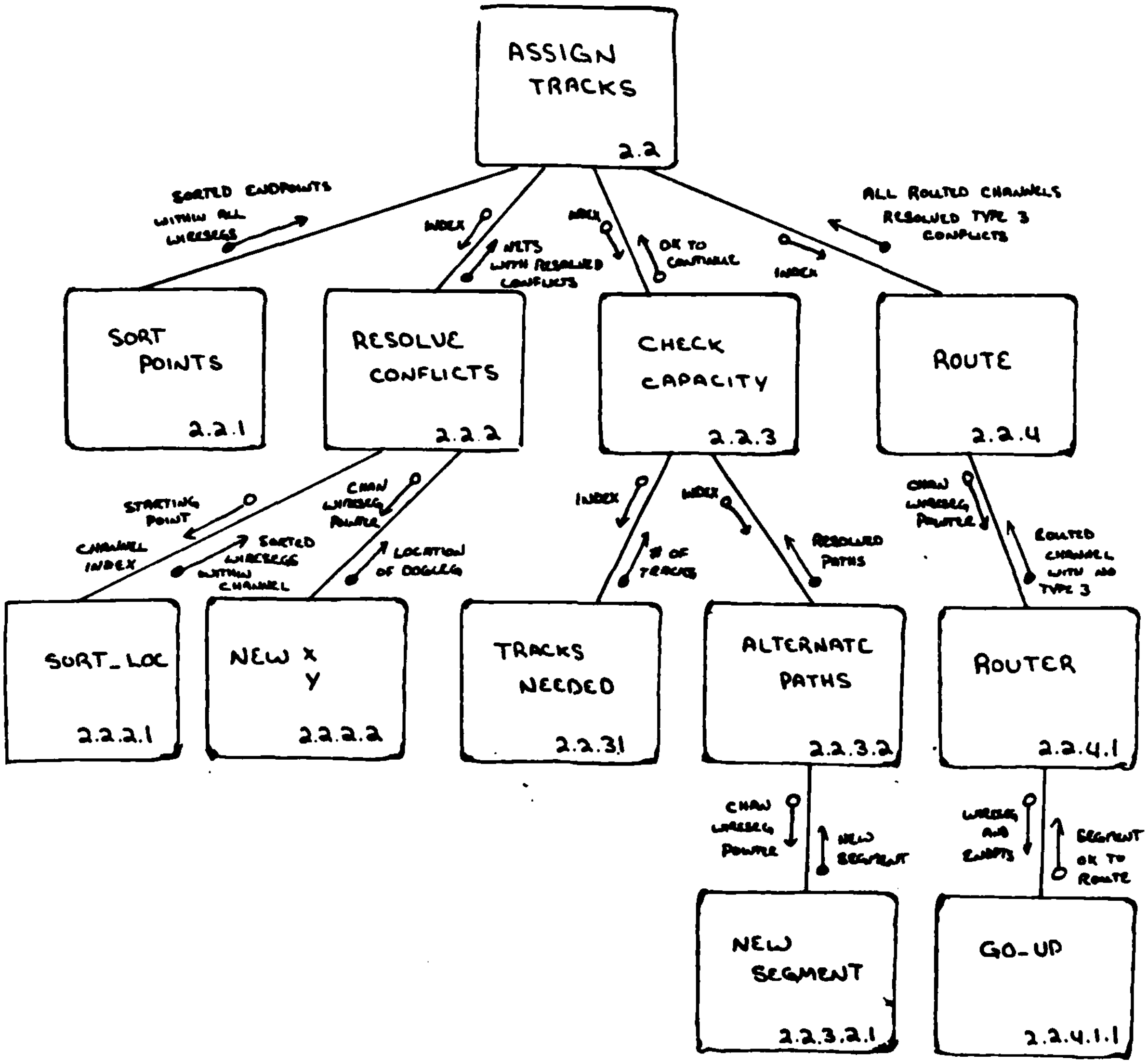



FIG. IV-3. THE ASSIGN TRACKS ROUTINES

Output: The wire segments that have specific track assignments.

Functions called: sort_points, resolve_conflicts, check_capacity, and route_channel

Calling functions: routing

Notes: See Figure IV-3

wire segments to be connected and the channel of the first segment.

Output: A new wire segment is added to try and complete the net.

Functions called: If the net is not complete horizontal calls vertical and vertical calls horizontal.

Calling functions: find_channel_path

Notes: A program adj pointers is called to adjust the pointers for the new wire segment.

Psuedo code:

```
               if two segments not connected and
               lie in the same channel
                 add segment to connect net
                 adjust pointers
   Horizontal  else
                 for all vertical channels
                   find closest vertical channel
                   add segment
                   call vertical

               if two segments not connected and
               lie in the same channel
                 add segment to connect net
                 adjust pointers
   Vertical    else
                 for all horizontal channels
                   find closest horizontal channel
                   add segment
                   call horizontal
```

Assign Tracks(2.2) This routine is responsible for accomplishing three tasks. The first task is to resolve Type 1 and Type 2 conflicts. The second task is to check tracks needed to route against tracks available. The final task is to resolve Type 3 conflicts and route the channels.

Input: The wire segments within the channel structures.

Direct Path(2.1.3.1) This routine completes a path between endpoints when they lie directly across from each other. If the endpoints are on the same layer and not on the routing layer for that channel, one wire segment will connect them. However, if the endpoints lie on the routing layer or if the endpoints lie on different layers, more than one segment may be necessary.

Input: The channel and net index are needed.

Output: New wire segment(s) are added to complete the routing path.

Functions called: None

Calling functions: find_channel_path

Notes: None

Psuedo code:

```
if endpoints lie on horizontal channel
  if wire segments meet in the center
    if wire segments on same layer
      connect net with first segment
    else
      change layers
      connect net
  else if start and end of net had to
                            change layers
    connect with three wire segments
  else
    connect with two wire segments
else
  repeat above only for vertical channels
```

Horizontal and Vertical(2.1.3.2 & 2.1.3.3) These two routines complete the path between endpoints of a net. They are called recursively until a path is complete.

Input: There are three inputs to these routines, the two

other cases. The second routine uses the following algorithm for finding a path.

1. Are the endpoints on the same channel? If yes, connect and stop.

2. Go to the closest opposite type of channel. If starting on horizontal go to closest vertical channel. Conversely, if starting on vertical go to closest horizontal channel.

3. Are the new endpoints on the same channel? If yes, connect and stop.

4. Go to closest opposite type of channel (see step 2).

5. Connect the endpoints.

Input: A list of nets found in the net array and the unconnected wire segments found in the channel structure.

Output: Wire segments are added to the channels to complete the path for a net.

Functions called: direct_path, horizontal, and vertical

Calling functions: assign_channels

Notes: This search algorithm works because all horizontal channels intersect all vertical channels and vice versa.

Psuedo code:

```
for all nets
  find two segments within net
                not connected
  if endpoints of segments lie across
                from each other then
    complete the path
  else
    alternate between horizontal and
    vertical channels until net complete
```

segments are perpendicular to the channel. For example, horizontal channels have vertical segments to their center. These segments can not be on the routing layer of a channel, metal for horizontal channels and metal2 for vertical channels. The perpendicular segments are stored in the <u>untrack</u> array and are not routed.

Input: An endpoint found in the net array and the address of the channel it is in.

Output: Wire segments are added to the channels.

Functions called: None

Calling functions: assign_channels

Notes: None

Psuedo code:

```
for all nets
  if net starts in vertical channel then
    if layer not = metal2 then
      path from start to center
    else
      change layers
      complete path to center
  else
    if layer not = metal then
      path from start to center
    else
      change layers
      complete path to center
```

<u>Find Channel Path</u>(2.1.3) This routine completes a path between the endpoints of a net. Track capacity of the channels is ignored, only that a path exists is important.

There are two routines that find a path for every net. The first routine finds a path when endpoints are directly across from each other. The second routine finds a path for

Psuedo code:

```
for all segments
  get a segment not routed
  if a segment to compare starts or ends
                at same location as this seg
    does the intersection of segments lie
                in a vertical channel?
    if yes, skip to next segment
    does the new segment need to be
                routed first?
    if yes, skip to next segment
assign and adjust pointers
```

<u>Go Up</u>(2.2.4.1.1) There are two routines that check to see if an unrouted wire segment connects to the top of the channel, one checks horizontal segments and one checks vertical segments.

Input: The three inputs to these routines are a pointer to the wire segment and the x and y coordinates of the end to check.

Output: If the segment connects upward then TRUE is returned, otherwise FALSE is returned.

Functions called: None

Calling functions: router

Notes: None

Psuedo code:

```
if the endpoint connects on the left
  if the leftend of the segment connects
                                to the endpoint
    if it connects upward
      return TRUE
    else
      return FALSE
  else
    if it connects upward
      return TRUE
    else
      return FALSE
      endpoint connects on the right
else
  (similar to the code above)
```

<u>Form CLL Statements</u>(2.3) The purpose of this routine is to form CLL statements that describe the routing path of the nets. The format of the output is a comment describing the net followed by the appropriate CLL WIRE and VIA statements that describe the routing path.

Input: The wire segments that make up a routing path for a net.

Output: The CLL output file.

Functions called: comment CLL

Calling functions: routing

Notes: See Figure IV-4

Psuedo code:

```
for all nets
  form a comment line
  form CLL WIRE and VIA statements
```

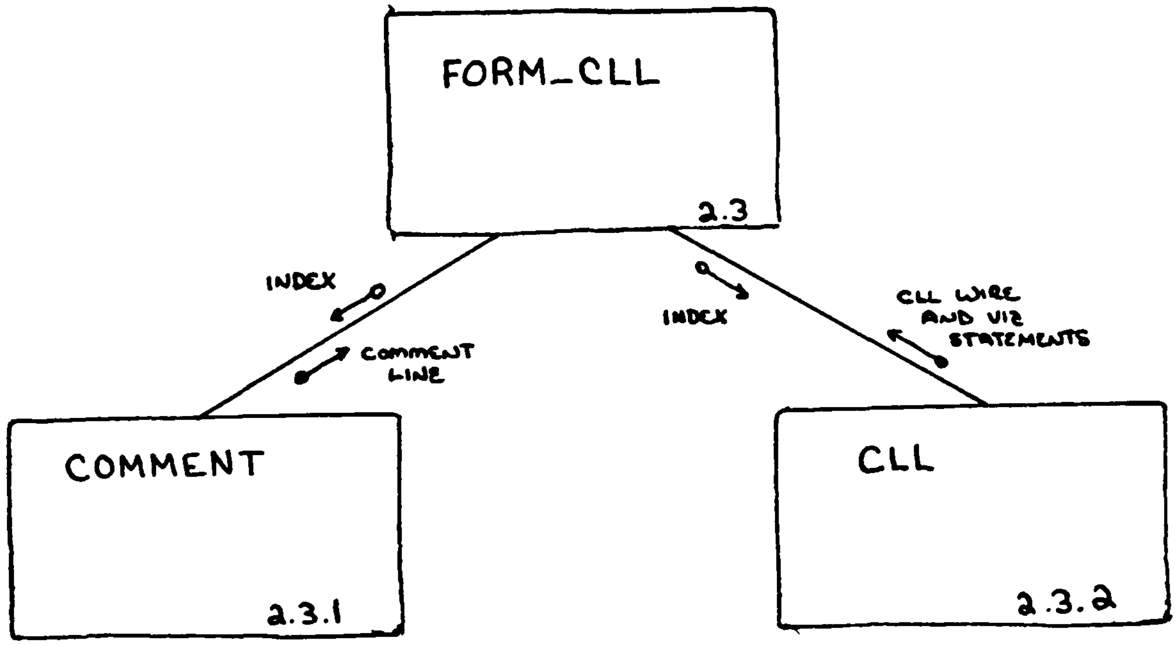



FIG. IV-4. THE FORM CLL ROUTINES

Comment(2.3.1) This routine builds a comment that describes a net. The comment is of the form:

"/* CONNECT x y layer to x y layer */"

Input: A pointer to a net.

Output: A comment line.

Functions called: None

Calling functions: form_CLL

Notes: None

Psuedo code:

```
find layer of start of the net
find layer of end of the net
form CONNECT statement comment line
```

CLL(2.3.2) This routine builds CLL WIRE and VIA statements that describe the path of a net.

Input: A pointer to a net.

Output: CLL WIRE and VIA statements that describe the path of a net.

Functions called: None

Calling functions: form_CLL

Notes: The CLL WIRE and VIA statements created by this program do not run on the local system. The metal2 layer has not been implemented yet. Any statement that includes the metal2 layer fails. Also the VIA statement needs a layer designator. It was left out because the local system may not allow a connection between metal and metal2 using a VIA statement.

Psuedo code:

```
find layer of starting layer
form WIRE statement
form VIA statement
if done STOP
while not done
  get a wire segment
  if last segment of a net
    find ending layer
    form wire statement
  else
    form WIRE statement
    locate VIA statement
    form VIA statement
```

# V. Conclusions

This chapter has four sections. First, a discussion of what the automatic routing program developed in this study can do. Second, a discussion of what the routing program can not do. Third, the automatic program is analyzed with respect to how well it accomplishes its goals. Finally, recommendations and closing comments are presented.

## What The Routing Program Can Do.

The automatic routing program developed in this study routes two point nets subject to certain constraints. The endpoints of a net must lie on or outside the channel boundary. Also, the channel must be wide enough to accomodate the routing paths of the nets. Another constraint is that a location exists for a dogleg if one is needed.

If a horizontal channel is not wide enough to route all of the wires assigned to it, an attempt is made to find an alternate path for some of the wire segments. To remove a horizontal wire segment, both endpoints must lie in a vertical channel. The wire segment is moved to a horizontal channel that has not been routed yet.

Contraint loops cause no problem. Type 1 and type 3 conflicts are taken care of by ordering the way the segments are routed. Type 2 conflicts are resolved by breaking a segment into two segments and introducing a dogleg.

## What The Routing Program Can Not Do.

The routing program does not accomplish complete automatic routing but only a limited subset. Multi-point nets greater than two are not routed. However, multi-point nets can be broken into two point nets or routed manually. Because power and ground connections are almost always multi-point nets, this constraint must be dealt with in all circuit designs.

Transistors created by poly wires crossing diffusion wires can not be specified. This condition was specifically avoided. Transistors must be manually routed.

The output of the routing program does not produce syntactically correct CLL statements. The VIA statements produced in the program do not include a layer designator. The layer designator was left off because the metal2 layer has not been implemented on the local system. There are numerous vias between metal and metal2 layers. Rather than guess how these two layers will be connected when implemented, it was purposely left out.

## Analysis.

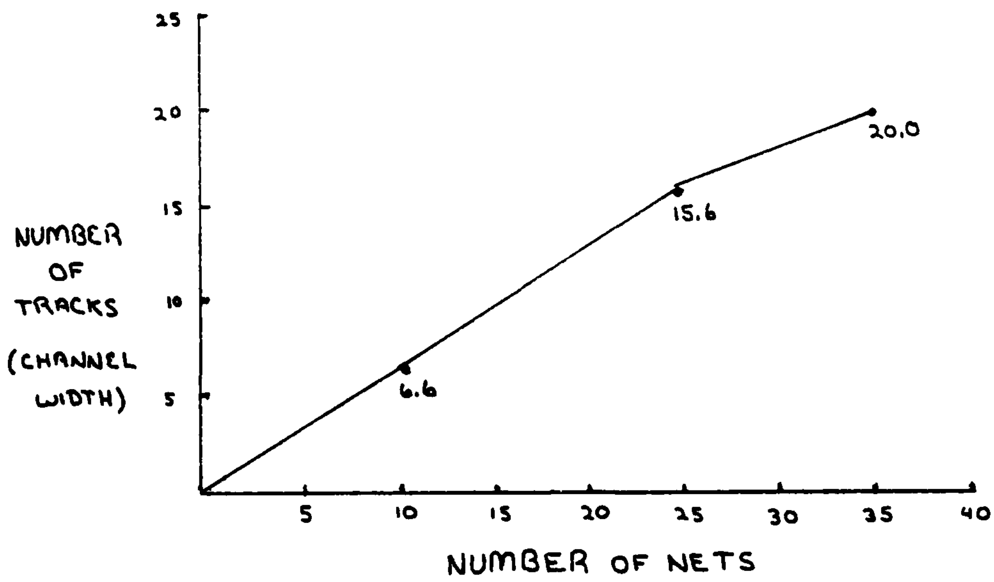
The routing program is a success if it meets its goals. The goal in this case was to automatically route the interconnections of a VLSI chip. This goal was met to satisfaction.

Advantages. While it is true that all interconnections can not be automatically routed, most can. The program can be used by students to simplify work on VLSI design projects.

The automatic routing program developed in this study routes nets very fast. 50 nets were connected in less than 1 second on a VAX 11/780. The routing program comes very close to being interactive. That is, because of the speed of the routing program, changes can be made quickly while at the terminal. Students can spend more time on design because the actual routing is done faster.

Plotting Output. To create a plot of the automatically routed wires, the output must be modified. Because the metal2 layer has not been implemented, all wires on the metal2 layer must be changed. Also, all VIA statements must be augmented with a layer designator. If metal2 is changed to metal and a global layer is added to the output, a plot can be created quickly. Although the output is not suited for final design, routing paths and VIA connections will be shown (see Appendix C).

Channel Width Analysis. Analysis was done to see how wide a channel must be to route 100%. Channels containing 10 nets, 25 nets, and 35 nets were analyzed. A small BASIC program was written to generate random nets. The channel was 500 units long. Five runs were made for 10 nets. Three runs were made for 25 and 35 nets. The results are shown below.




It can be seen from the results that the channel width must be roughly two thirds the number of nets in the channel. However, this is just an estimate.

Recommendations.

There are several recommendations to be made concerning the work done in this thesis.

1. The metal2 layer must be implemented in the local version of CLL.

2. The code must be added to the automatic routing program to connect points between metal and metal2 layers.

3. The scope of the routing program should be expanded to include multi-point nets and transistors.

4. Channel descriptions should be relaxed so that channels are not limited to rectangular shapes.

5. A new search algorithm to connect endpoints would allow horizontal and vertical channels the freedom not to intersect.

6. The routing program should be modified to find alternate paths for wires within vertical channels.

7. The routing program should eventually be merged with the CLL program. Channel descriptions could be calculated from the cell placement directly.

In general, an attempt should be made to relax or remove all of the restrictions that were imposed on the routing program.

## Bibliography

Akers, Sheldon B. "Routing" in Design Automation Of Digital Systems, Volume 1, Theory and Techniques, edited by Melvin A. Breuer. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1972.

Breuer, Melvin A. "Recent Developments in the Automated Design and Analysis of Digital Systems," Proceedings Of The IEEE, Vol. 60, No. 1, January 1972. 12-27. New York: Institute of Electronics Engineers, Inc., 1972.

Deutsch, David N. "A "Dogleg" Channel Router," 13th Design Automation Conference Proceedings. 425-433. New York: Institute of Electronics Engineers, Inc., 1976.

Hashimoto, Akihiro and James Stevens. "Wire Routing By Optimizing Channel Assignment Within Large Apertures," 8th Design Automation Workshop Proceedings. 155-169. Atlantic City: Institute of Electronics Engineers, Inc., 1971.

Hightower, David W. "The Interconnection Problem - A Tutorial," 10th Design Automation Workshop Proceedings. 1-21. New York: Institute of Electronics Engineers, Inc., 1973.

Lee, C. Y. "An Algorithm for Path Connections and Its Applications," IRE Transactions on Electronic Computers. 346-365. Institute of Radio Engineers, 1961.

Mead, Carver A. and Lynn Conway. Introduction to VLSI Systems. Philippines: Addison-Wesley Publishing Company, 1980.

Soukup, Jiri. "Circuit Layout," Proceedings Of The IEEE, Vol. 69, No. 10, October 1981. 1281-1304. New York: Institute of Electronics Engineers, Inc., 1981.

## Appendix A:

## Sample Input to Automatic Routing Program

Sample input on next page.

```
BEGIN_LAYER metal2,metal,poly,diff END_LAYER
BEGIN_HCHANNELS 0,0              1100,100
0,245            1100,350
0,495            1100,600
0,745            1100,850
0,995            1100,1100
0,1245           1100,1350 END_CHANNELS
BEGIN_VCHANNELS 0,0              100,1350
200,0            300,1350
400,0            500,1350
600,0            700,1350
800,0            900,1350
1000,0           1100,1350 END_CHANNELS
CONNECT          150,245,poly            750,1245,poly
CONNECT          350,245,metal           550,995,diff
CONNECT          550,245,metal           600,948,poly
CONNECT          750,245,diff            600,854,diff
CONNECT          950,245,poly            500,948,poly
CONNECT          150,350,metal           950,350,diff
CONNECT          350,350,diff            1000,1198,diff
CONNECT          550,350,poly            100,948,poly
CONNECT          750,350,poly            150,995,metal
CONNECT          350,850,metal           750,745,metal
CONNECT          350,745,diff            750,850,poly
CONNECT          300,1115,poly           400,1115,diff
CONNECT          300,1125,diff           400,1125,diff
CONNECT          300,1135,metal          400,1135,metal2
CONNECT          507,600,poly            515,600,poly
CONNECT          507,495,diff            515,495,metal
CONNECT          105,495,poly            950,600,poly
CONNECT          105,600,diff            950,495,poly
CONNECT          920,1100,metal          110,745,metal
CONNECT          940,1100,metal2         120,745,metal2
CONNECT          960,1100,diff           130,745,poly
CONNECT          800,110,poly            800,1160,metal
CONNECT          800,120,diff            900 125 diff
CONNECT          800,130,poly            900,605,poly
CONNECT          800,605,diff            900,130,metal
CONNECT          800,140,diff            800,860,metal2
CONNECT          900,140,metal           900,150,diff
CONNECT          800,375,diff            800,875,poly
CONNECT          200,375,poly            300,875,diff
CONNECT          200,875,diff            300,375,poly
CONNECT          150,1245,poly           1000,610,diff
CONNECT          160,1245,diff           1000,630,metal
CONNECT          170,1245,metal2         1000,650,metal2
```

Appendix B:

Sample Output to Automatic
Routing Program

Sample output on next page.

```
#include        "/usr/lib/local/s_ext.cll"

sample
{
iterate 5,5     200,250
        NOut8(100,100);
poly;
/* CONNECT 150,245 poly to 750,1245 poly */
wire  poly    150,245      150,317;
via  148,315;
wire  metal     150,317      686,317;
via  684,315;
wire  metal     686,317      686,1322;
via  684,1320;
wire  metal     686,1322      750,1322;
via  748,1320;
wire  poly     750,1322     750,1245;


/* CONNECT 350,245 metal to 550,995 diff */
wire  metal    350,245      350,250;
via  348,248;
wire  metal     350,250      350,303;
via  348,301;
wire  metal     350,303      486,303;
via  484,301;
wire  metal     486,303      486,1046;
via  484,1044;
wire  metal     486,1046      550,1046;
via  548,1044;
wire  diff     550,1046     550,995;


/* CONNECT 550,245 metal to 600,948 poly */
wire  metal    550,245      550,250;
via  548,248;
wire  metal     550,250      550,303;
via  548,301;
wire  metal     550,303      693,303;
via  691,301;
wire  metal     693,303      693,948;
via  691,946;
wire  poly     693,948      600,948;


/* CONNECT 750,245 diff to 600,854 diff */
wire  diff    750,245      750,331;
via  748,329;
wire  metal     679,331      750,331;
via  677,329;
wire  metal     679,331      679,854;
via  677,852;
wire  diff     679,854      600,854;


/* CONNECT 950,245 poly to 500,948 poly */
wire  poly    950,245      950,296;
via  948,294;
wire  metal     493,296      950,296;
via  491,294;
wire  metal     493,296      493,948;
via  491,946;
wire  poly     493,948      500,948;
```

```
mmetal  = 0;   /* turn off default layers */
mmetal2 = 0;
ppoly   = 0;
ddiff   = 0;

flag = 0;

getword();

while ((a = strcmp(LAYEREND, buf)) != 0)
	{
	if ((a = strcmp(METAL, buf)) == 0)
		mmetal = 1;
	else
		if ((a = strcmp(POLY, buf)) == 0)
			ppoly = 1;
		else
			if ((a = strcmp(DIFF, buf)) == 0)
				ddiff = 1;
			else
				if ((a = strcmp(METAL2, buf)) == 0)
					mmetal2 = 1;
				else
					if ((a = strcmp(POLY2, buf)) == 0)
						ppoly2 = 1;
					else
					   error("illegal layer",buf);
	getword();
	}
if ((a = mmetal + mmetal2) <  2)
	error("missing required layers",NULL);
}


hchannel_input()
/*****************************************************************

FUNCTION:	hchannel_input

PURPOSE:	This routine processes horizontal channel input.
	The corners of the channel are input and the center is
	calculated.

*****************************************************************/
{
int	a;

getword();
while ((a = strcmp(ENDCHNL, buf)) != 0)
	{
	hchan[lhchan].corner.xloc = ston();
	getword();
	hchan[lhchan].corner.yloc = ston();
```

```
}

initializing()
/****************************************************************

FUNCTION:   initializing

PURPOSE:    This routine searches a file and depending on
        the keyword found layer, channel, or net input is
        processed.

****************************************************************/

{
int     a;

while (!noinput)      /* while there is input to process */
        {
        getword();    /* get the next word */

        if (noinput)  /* if EOF stop processing */
                return;

        if ((a = strcmp(LAYER, buf)) == 0)  /* is keyword layer? */
                layer_input();
        else
                if ((a = strcmp(HCHANNL, buf)) == 0)      /* is it a horizontal */
                        hchannel_input();           /* channel?         */
                else
                if ((a = strcmp(VCHANNL, buf)) == 0)      /* is it a vertical  */
                        vchannel_input();           /* channel?        */
                else
                        if ((a = strcmp(NET, buf)) == 0) /* is it a net      */
                                net_input();
                        else
                                error("illegal input",buf);
        }
}

layer_input()
/****************************************************************

FUNCTION:   layer_input

PURPOSE:    This routine processes layer input.  It turns off
        all default layers and then turns on specified layers.
        It also checks to see if metal and metal2 are present.

****************************************************************/
{
int     a,flag;
```

The "word" is then stored in a buffer called buf.

```
******************************************************************/
{
char    c;
int     indx;

indx = 0;

c = getchar();                  /* get a character and if EOF */
if (c == EOF)                   /* return                      */
        {
        noinput = TRUE;
        return;
        }

/* skip while character is equal to blanks, newlines, tabs, or commas */
while ((c == ' ') || (c == '\n') || (c == '\t') || (c == ','))
        c = getchar();

buf[indx] = c;          /* store first character into buf       */
c = getchar();

/* while character not equal to blanks, newlines, tabs, or commas       */
/* store character into buf                                      */
while ((c != ' ') && (c != '\n') && (c != '\t') && (c != ','))
        {
        indx++;
        buf[indx] = c;
        c = getchar();
        }

buf[++indx] = '\0';     /* attach a NULL character to the end of buf */
}


int     ston()
/******************************************************************

FUNCTION:     ston    string to number

PURPOSE:      This routine converts a character string found
        in buf to a number.

******************************************************************/
{
int     i, num;

num = 0;

for (i = 0; buf[i] != '\0'; i++)
        num = num * 10 + (buf[i] - '0');

return (num);
```

*ainit.c*

```
#include      "auto.h"
main ()
/*****************************************************************

FUNCTION:     main

PURPOSE:      This is the main routine. It initializes a few
       variables and calls the initializing and the routing
       routines.

*****************************************************************/
{
int    i;

mmetal  = 1;  /* turn on default layers */
mmetal2 = 1;
ppoly   = 1;
ddiff   = 1;

for (i=0; i < VCHNMAX; i++) /* a 1 in the id field of the channel */
       vchan[i].id = 1;       /* denotes a vertical channel     */

initializing();          /* call the initializing routines */

routing();               /* call the routing routines */

exit(0);

}


error(s1, s2)
/*****************************************************************

FUNCTION:     error

PURPOSE:      This routine prints two character strings and halts
       the program.

*****************************************************************/
char    *s1, *s2;
{
printf("%s %s\n",s1, s2);
exit(1);
}


getword()
/*****************************************************************

FUNCTION:     getword

PURPOSE:      This routine scans an input file for a "word"
       that is seperated by blanks, tabs, commas, or end-of-line
```

```
        int     yloc;
};

struct wireseg {
        int     tag;
struct point  leftend;
struct point  rightend;
struct wireseg         *left;
struct wireseg *right;
};

struct channel {
        int     id;
        int     done;
        int     center;
        int     luseg;
        int     ltseg;
struct point  corner;
struct point  opcorner;
struct wireseg         track[TRKSEGS];
struct wireseg untrack[UNTRKSG];
} hchan[HCHNMAX], vchan[VCHNMAX];

struct net {
        char    layer[2];
struct point  start;
struct point  end;
struct wireseg *wpoint;
struct channel         *pointer[2];
} nets[NUMNETS];
```

*auto.h*

```
#include      "stdio.h"

#define          TRKSEGS      50          /* max # of tracked wire segments */
#define          UNTRKSG      100         /* max # of untracked wire segments */
#define          HCHNMAX      10          /* max # of horizontal channels */
#define          VCHNMAX      10          /* max # of vertical channels */
#define          NUMNETS      100         /* max # of nets */
#define          MINDIST 7           /* distance between tracks and endpts */

#define          METAL "metal"       /* layers that can be used */
#define          POLY  "poly"
#define          DIFF  "diff"
#define          METAL2      "metal2"
#define          POLY2 "poly2"

#define          NULL  0
#define          BUFSIZE      30          /* max size of word array buf */
#define          TRUE  1
#define          FALSE 0

#define          NET    "CONNECT"     /* delimiter for net statement */

#define          LAYER "BEGIN_LAYER"       /* delimiter for layer statement */
#define          LAYEREND "END_LAYER"

#define          HCHANNL     "BEGIN_HCHANNELS"  /* delimiter for channel statement */
#define          VCHANNL     "BEGIN_VCHANNELS"
#define          ENDCHNL     "END_CHANNELS"

/****************************************

  GLOBAL VARIABLES

****************************************/

char        buf[BUFSIZE];        /* word array buf */
int         s_array[TRKSEGS];    /* sort array for wire segments */

int         mmetal;   /* metal layer flag */
int         ppoly;    /* poly layer flag */
int         ddiff;    /* dif layer flag */
int         mmetal2;  /* metal2 layer flag */
int         ppoly2;   /* poly2 layer flag */
int         noinput;  /* end of input flag */
int         lhchan;   /* last horizontal channel */
int         lvchan;          /* last vertical channel */
int         lnet;     /* last net */
int         top;      /* top of the channel */
int         bottom;   /* bottom of the channel */
int         resetptr; /* start of the channel */
int         trackptr; /* pointer to a track */

struct point {
      int     xloc;
```

## Appendix D:

## Source Code for Automatic Routing Program

Source code on next page.


747
bounds

# Appendix C:

## CLL Plot Using Automatic Routing Program

CLL plot on next page.

```
/* CONNECT 200,875 diff to 300,375 poly */
wire  diff    200,875     286,875;
via  284,873;
wire  metal      286,375     286,875;
via  284,373;
wire  poly     286,375     300,375;

/* CONNECT 150,1245 poly to 1000,610 diff */
wire  poly    150,1245     150,1343;
via  148,1341;
wire  metal      150,1343     1086,1343;
via  1084,1341;
wire  metal      1086,610     1086,1343;
via  1084,608;
wire  diff      1086,610     1000,610;

/* CONNECT 160,1245 diff to 1000,630 metal */
wire  diff    160,1245     160,1336;
via  158,1334;
wire  metal      160,1336     1079,1336;
via  1077,1334;
wire  metal      1079,630     1079,1336;
via  1077,628;
wire  metal      1079,630     1000,630;

/* CONNECT 170,1245 metal to 1000,650 metal2 */
wire  metal    170,1245     170,1329;
via  168,1327;
wire  metal      170,1329     1072,1329;
via  1070,1327;
wire  metal      1072,650     1072,1329;
via  1070,648;
wire  metal      1072,650     1005,650;
via  1003,648;
wire  metal      1005,650     1000,650;
}
```

```
/* CONNECT 800,120 diff to 900,125 diff */
wire  diff     800,120     886,120;
via  884,118;
wire  metal      886,120     886,125;
via  884,123;
wire  diff      886,125     900,125;

/* CONNECT 800,130 poly to 900,605 poly */
wire  poly     800,130     865,130;
via  863,128;
wire  metal      865,130     865,150;
via  863,148;
wire  metal      879,150     865,150;
via  877,148;
wire  metal      879,150     879,605;
via  877,603;
wire  poly      879,605     900,605;

/* CONNECT 800,605 diff to 900,130 metal */
wire  diff     800,605     872,605;
via  870,603;
wire  metal      872,130     872,605;
via  870,128;
wire  metal      872,130     900,130;

/* CONNECT 800,140 diff to 800,860 metal */
wire  diff     800,140     858,140;
via  856,138;
wire  metal      858,140     858,860;
via  856,858;
wire  metal      858,860     805,860;
via  803,858;
wire  metal      805,860     800,860;

/* CONNECT 900,140 metal to 900,150 diff */
wire  metal     900,140     886,140;
via  884,138;
wire  metal      886,140     886,150;
via  884,148;
wire  diff      886,150     900,150;

/* CONNECT 800,375 diff to 800,875 poly */
wire  diff     800,375     886,375;
via  884,373;
wire  metal      886,375     886,875;
via  884,873;
wire  poly      886,875     800,875;

/* CONNECT 200,375 poly to 300,875 diff */
wire  poly     200,375     279,375;
via  277,373;
wire  metal      279,375     279,385;
via  277,383;
wire  metal      293,385     279,385;
via  291,383;
wire  metal      293,385     293,875;
via  291,873;
wire  diff      293,875     300,875;
```

```
via  113,577;
wire  metal     115,593     115,579;
via  113,591;
wire  metal     115,593     950,593;
via  948,591;
wire  poly     950,593     950,600;

/* CONNECT 105,600 diff to 950,495 poly */
wire  diff     105,600     105,586;
via  103,584;
wire  metal     105,586     950,586;
via  948,584;
wire  poly     950,586     950,495;

/* CONNECT 920,1100 metal to 110,745 metal */
wire  metal    920,1100     920,1095;
via  918,1093;
wire  metal     920,1095     920,1074;
via  918,1072;
wire  metal     65,1074     920,1074;
via  63,1072;
wire  metal     65,838     65,1074;
via  63,836;
wire  metal     65,838     110,838;
via  108,836;
wire  metal     110,838     110,750;
via  108,748;
wire  metal     110,750     110,745;

/* CONNECT 940,1100 metal to 120,745 metal2 */
wire  metal    940,1100     940,1081;
via  938,1079;
wire  metal     72,1081     940,1081;
via  70,1079;
wire  metal     72,831     72,1081;
via  70,829;
wire  metal     72,831     120,831;
via  118,829;
wire  metal     120,831     120,745;

/* CONNECT 960,1100 diff to 130,745 poly */
wire  diff     960,1100     960,1088;
via  958,1086;
wire  metal     79,1088     960,1088;
via  77,1086;
wire  metal     79,824     79,1088;
via  77,822;
wire  metal     79,824     130,824;
via  128,822;
wire  poly     130,824     130,745;

/* CONNECT 800,110 poly to 800,1160 metal */
wire  poly     800,110     893,110;
via  891,108;
wire  metal     893,110     893,1160;
via  891,1158;
wire  metal     893,1160     800,1160;
```

```
via  748,836;
wire  poly     750,838     750,850;

/* CONNECT 300,1115 poly to 400,1115 diff */
wire  poly    300,1115     293,1115;
via  291,1113;
wire  metal     293,1053     293,1115;
via  291,1051;
wire  metal     293,1053     493,1053;
via  491,1051;
wire  metal     493,1053     493,1115;
via  491,1113;
wire  diff     493,1115     400,1115;

/* CONNECT 300,1125 diff to 400,1125 diff */
wire  diff     300,1125     279,1125;
via  277,1123;
wire  metal     279,1067     279,1125;
via  277,1065;
wire  metal     279,1067     479,1067;
via  477,1065;
wire  metal     479,1067     479,1125;
via  477,1123;
wire  diff     479,1125     400,1125;

/* CONNECT 300,1135 metal to 400,1135 metal */
wire  metal     300,1135     286,1135;
via  284,1133;
wire  metal     286,1060     286,1135;
via  284,1058;
wire  metal     286,1060     486,1060;
via  484,1058;
wire  metal     486,1060     486,1135;
via  484,1133;
wire  metal     486,1135     405,1135;
via  403,1133;
wire  metal     405,1135     400,1135;

/* CONNECT 507,600 poly to 515,600 poly */
wire  poly    507,600     507,579;
via  505,577;
wire  metal     507,579     515,579;
via  513,577;
wire  poly     515,579     515,600;

/* CONNECT 507,495 diff to 515,495 metal */
wire  diff    507,495     507,572;
via  505,570;
wire  metal     507,572     515,572;
via  513,570;
wire  metal     515,572     515,500;
via  513,498;
wire  metal     515,500     515,495;

/* CONNECT 105,495 poly to 950,600 poly */
wire  poly    105,495     105,579;
via  103,577;
wire  metal     105,579     115,579;
```

```
/* CONNECT 150,350 metal to 950,350 diff */
wire  metal     150,350      150,345;
via  148,343;
wire  metal      150,345      150,324;
via  148,322;
wire  metal      150,324      950,324;
via  948,322;
wire  diff      950,324      950,350;


/* CONNECT 350,350 diff to 1000,1198 diff */
wire  diff     350,350      350,310;
via  348,308;
wire  metal      350,310      1093,310;
via  1091,308;
wire  metal      1093,310      1093,1198;
via  1091,1196;
wire  diff      1093,1198      1000,1198;


/* CONNECT 550,350 poly to 100,948 poly */
wire  poly     550,350      550,331;
via  548,329;
wire  metal      93,331      550,331;
via  91,329;
wire  metal      93,331      93,948;
via  91,946;
wire  poly      93,948      100,948;


/* CONNECT 750,350 poly to 150,995 metal */
wire  poly     750,350      750,338;
via  748,336;
wire  metal      86,338      750,338;
via  84,336;
wire  metal      86,338      86,1067;
via  84,1065;
wire  metal      86,1067      150,1067;
via  148,1065;
wire  metal      150,1067      150,1000;
via  148,998;
wire  metal      150,1000      150,995;


/* CONNECT 350,850 metal to 750,745 metal */
wire  metal     350,850      350,845;
via  348,843;
wire  metal      350,845      350,831;
via  348,829;
wire  metal      350,831      750,831;
via  748,829;
wire  metal      750,831      750,750;
via  748,748;
wire  metal      750,750      750,745;


/* CONNECT 350,745 diff to 750,850 poly */
wire  diff     350,745      350,824;
via  348,822;
wire  metal      350,824      360,824;
via  358,822;
wire  metal      360,838      360,824;
via  358,836;
wire  metal      360,838      750,838;
```

```
        getword();
        hchan[lhchan].opcorner.xloc = ston();
        getword();
        hchan[lhchan].opcorner.yloc = ston();
        lhchan++;
        getword();
        }

/* compute the center of each channel */
for (a = 0; a <= (lhchan - 1); a++)
        hchan[a].center = (hchan[a].corner.yloc + hchan[a].opcorner.yloc) / 2;
}


vchannel_input()
/*****************************************************************

FUNCTION:     vchannel_input

PURPOSE:      This routine processes vertical channel input.
        The corners of the channel are input and the center is
        calculated.

*****************************************************************/
{
int     a;

getword();
while ((a = strcmp(ENDCHNL, buf)) != 0)
        {
        vchan[lvchan].corner.xloc = ston();
        getword();
        vchan[lvchan].corner.yloc = ston();
        getword();
        vchan[lvchan].opcorner.xloc = ston();
        getword();
        vchan[lvchan].opcorner.yloc = ston();
        lvchan++;
        getword();
        }

for (a = 0; a <= (lvchan - 1); a++)
        vchan[a].center = (vchan[a].corner.xloc + vchan[a].opcorner.xloc) / 2;
}

char    layers()
/*****************************************************************

FUNCTION:     layers

PURPOSE:      This routine stores a character in the net
        structure defining what layer the endpoint is on.  A
        check is made first to see if it is a valid layer.
```

```
*****************************************************************/
{
int     a;

        if ((a = strcmp(METAL, buf)) == 0)
                return('m');
        else
                if ((a = strcmp(METAL2, buf)) == 0)
                        return('2');
                else
                        if ((a = strcmp(DIFF, buf)) == 0)
                                if (ddiff == 0)
                                  error("diff layer not available",NULL);
                                else
                                  return('d');
                else
                        if ((a = strcmp(POLY, buf)) == 0)
                                if (ppoly == 0)
                                  error("poly layer not available",NULL);
                                else
                                  return('p');
                else
                        if ((a = strcmp(POLY2, buf)) == 0)
                                if (ppoly2 == 0)
                                  error("poly2 layer not available\n",NULL);
                                else
                                  return('P');

                else error("not a valid layer",buf);
}

net_input()
/*****************************************************************

FUNCTION:       net_input

PURPOSE:        This routine fills in the net structure for each
        net.  The x and y locations for the end points and the
        layer designation is stored for each net.

*****************************************************************/
{
getword();
nets[lnet].start.xloc = ston();

getword();
nets[lnet].start.yloc = ston();                 /* the starting point of the net */

getword();
nets[lnet].layer[0] = layers();

getword();
nets[lnet].end.xloc = ston();
```

```
getword();
nets[lnet].end.yloc = ston();          /* the ending point of the net */

getword();
nets[lnet].layer[1] = layers();

lnet++;

}
```

*aroute.c*

```
#include        "auto.h"
routing()
/*************************************************************************

FUNCTION:    routing

PURPOSE:     The purpose of this routine is to call the routing
       routines.

*************************************************************************/
{
assign_channels();     /* this routine finds a routing path for a net */

assign_tracks();       /* this routine finds a specific track for the path */

form_cll();            /* this routine converts the path into CLL statements */
printy();
}

assign_channels()
/*************************************************************************

FUNCTION:    assign_channels

PURPOSE:     The purpose of this routine is to find a routing path for
       a net. To do this the channels that the end points are in must
       be found. Next, a segment from the endpoint to the center of its
       channel is found. And finally, the endpoints are connected by a
       routing algorithm.

*************************************************************************/
{
find_start_chan();     /* find the starting channel */

find_end_chan();       /* find the ending channel  */

center_start();        /* segment from start point to center of channel */

center_end();          /* segment from end point to center of channel  */

find_channel_path(); /* connect the endpoints    */
}

find_start_chan()
/*************************************************************************

FUNCTION:    find_start_chan

PURPOSE:     The purpose of this routine is to find out which channel
       the start of the net is in. The horizontal channels are searched
       first and the vertical channels are searched. If the endpoint
       lies on a channel boundary the channel is found. If the endpoint
       lies off the channel boundary the closest channel to the endpoint
       is used.
```

```
*************************************************************************/
{
int     i,ii,netpt,ch1pt,ch2pt,flag,delta;
int     smallest;

for (i = 0; i < lnet; i++)          /* for all of the nets find the */
        {                           /* starting point channel       */

        delta = 0;                  /* delta holds the current difference */
        smallest = 1000;            /* smallest holds the smallest diff   */
        flag = FALSE;               /* flag = TRUE if starting channel found */

        netpt = nets[i].start.yloc;      /* netpt is the starting point */

        for (ii = 0; ii < lhchan; ii++) /* for all of the horiz channels */
                {
                ch1pt = hchan[ii].corner.yloc;       /* channel boundary */
                ch2pt = hchan[ii].opcorner.yloc;

                delta = small(netpt, ch1pt, ch2pt); /* returns a difference */
                                                    /* from the channel     */
                if (delta == 0) /* if 0 channel found */
                        {
                        nets[i].pointer[0] = &hchan[ii];
                        flag = TRUE;
                        break;
                        }
                else
                        if (delta < smallest)  /* is difference smaller than */
                                {           /* the smallest difference    */
                                nets[i].pointer[0] = &hchan[ii];
                                smallest = delta;
                                }
                }
        if (!flag)         /* if channel has not been found yet */
          {
          netpt = nets[i].start.xloc;  /* netpt is the starting point */

          for (ii = 0; ii < lvchan; ii++)  /* for all vertical channels */
                {
                ch1pt = vchan[ii].corner.xloc;       /* channel boundarys */
                ch2pt = vchan[ii].opcorner.xloc;

                delta = small(netpt, ch1pt, ch2pt); /* returns difference */
                                                   /* from the channel   */
                if (delta == 0) /* if 0 channel found */
                        {
                        nets[i].pointer[0] = &vchan[ii];
                        break;
                        }
                else
                        if (delta < smallest)  /* is difference smaller than */
                                {             /* smallest difference       */
```

```
                                        nets[i].pointer[0] = &vchan[ii];
                                        smallest = delta;
                                        }
                          }
                     }
                }
}

find_end_chan()
/*************************************************************************

FUNCTION:    find_end_chan

PURPOSE:     The purpose of this routine is to find out which channel
        the end of the net is in. The horizontal channels are searched
        first and the vertical channels are searched. If the endpoint
        lies on a channel boundary the channel is found. If the endpoint
        lies off the channel boundary the closest channel to the endpoint
        is used.

*************************************************************************/
{
int     i,ii,netpt,ch1pt,ch2pt,flag,delta;
int     smallest;

for (i = 0; i < lnet; i++)          /* for all nets find the channel of */
        {                           /* the end point                   */

        delta = 0;                  /* delta is the current difference */
        smallest = 1000;            /* smallest is smallest difference */
        flag = FALSE;               /* flag = TRUE if channel found    */

        netpt = nets[i].end.yloc;        /* netpt holds the end point */

        for (ii = 0; ii < lhchan; ii++) /* for all horizontal channels */
                {
                ch1pt = hchan[ii].corner.yloc;       /* channel boundarys */
                ch2pt = hchan[ii].opcorner.yloc;

                delta = small(netpt, ch1pt, ch2pt);  /* returns difference */
                                                     /* from channel      */
                if (delta == 0) /* if 0 channel found */
                        {
                        nets[i].pointer[1] = &hchan[ii];
                        flag = TRUE;
                        break;
                        }
                else
                        if (delta < smallest)  /* is difference smaller than */
                                {              /* smallest difference      */
                                nets[i].pointer[1] = &hchan[ii];
                                smallest = delta;
                                }
                }
```

```
        if (!flag)      /* if channel has not been found */
          {
          netpt = nets[i].end.xloc;   /* netpt contains the end point */

          for (ii = 0; ii < lvchan; ii++)        /* for all vertical channels */
               {
               ch1pt = vchan[ii].corner.xloc;  /* channel boundarys */
               ch2pt = vchan[ii].opcorner.xloc;

               delta = small(netpt, ch1pt, ch2pt);  /* returns difference */
                                                    /* from channel      */
               if (delta == 0) /* if 0 channel found */
                      {
                      nets[i].pointer[1] = &vchan[ii];
                      break;
                      }
               else
                      if (delta < smallest)  /* is difference smaller than */
                             {              /* smallest difference        */
                             nets[i].pointer[1] = &vchan[ii];
                             smallest = delta;
                             }
               }
          }
        }
}

int     small(netpt, ch1pt, ch2pt)
int     netpt, ch1pt, ch2pt;
/***********************************************************************

FUNCTION:     small

PURPOSE:      The purpose of this routine is to find out how far netpt
        is from the channel described by ch1pt and ch2pt.  The difference
        is returned.

***********************************************************************/
{
int     delta;

if (ch1pt < ch2pt)    /* if the 1st point is smaller than 2nd        */

        if (netpt < = ch1pt)   /* the netpt lies below the channel */
               delta = ch1pt - netpt;
        else                   /* the netpt lies above the channel */
               delta = netpt - ch2pt;

else                   /* the 2nd point is smaller than the 1st */

        if (netpt < = ch2pt)   /* the netpt lies below the channel */
               delta = ch2pt - netpt;
        else                   /* the netpt lies above the channel */
               delta = netpt - ch1pt;
```

```
return (delta);              /* return the difference */
}

find_channel_path()
/*************************************************************************

FUNCTION:    find_channel_path

PURPOSE:     The purpose of this routine is to connect the endpoints
       of a net. The path found travels down the center of the channel.

*************************************************************************/
{
struct wireseg      *w1;
struct wireseg *w2;
struct channel *c;

int    i;

for (i=0; i < lnet; i++)        /* for all nets find a path */
       {
       w1 = nets[i].wpoint;     /* w1 contains starting wireseg address */
       w2 = nets[i].wpoint->right; /* w2 contains ending wireseg address  */
       c = nets[i].pointer[0];    /* c contains the starting channel addr */

       /* while the segment is not unbroken, that is no holes in the path */
       while ((w1->rightend.xloc == w2->leftend.xloc) &&
           (w1->rightend.yloc == w2->leftend.yloc))
         if (w2->right == NULL)
              break;
         else
              {
              w1 = w2;            /* w1 contains one side of the break */
              w2 = w1->right;          /* w2 contains other side of break  */
              }

       if ((w1->rightend.xloc == w2->leftend.xloc) &&
           (w1->rightend.yloc == w2->leftend.yloc))
              direct_path(c,i);   /* endpoints lie across from each other */
       else
        if (c->id == 1)       /* if the starting channel is a vertical one */
              vertical(w1, w2, c);
        else
              horizontal(w1, w2, c);
       }
}

direct_path(c, i)
struct channel *c;
       int     i;
/*************************************************************************

FUNCTION:    direct_path
```

PURPOSE: The purpose of this routine is to find a path for a net when the endpoints lie directly across from each other. There are 4 possibilites.

```
          type1: neither endpoint lies on the routing layer
    type2 type3: one of the endpoints lie on the routing layer
          type4: both of the endpoints lie on the routing layer

*************************************************************************/
{
struct wireseg *w1,*w2;

w1 = nets[i].wpoint; /* first wire seg in net */
w2 = w1-> right;          /* next wire seg in net  */

if (c-> id == 0)              /* horizontal channel   */
      if (w1-> rightend.yloc == c-> center)       /* type 1 or 3 */
            if (w2-> rightend.yloc == nets[i].end.yloc)       /* type 1 */
                  if (nets[i].layer[0] == nets[i].layer[1])
                    {
                    w1-> rightend.yloc = nets[i].end.yloc;
                    w1-> right = NULL;
                    }
                  else
                   if (nets[i].start.yloc <  nets[i].end.yloc)
                     {
                     w1-> rightend.yloc = nets[i].end.yloc - 5;
                     w2-> leftend.yloc = w1-> rightend.yloc;
                     }
                   else
                     {
                     w1-> rightend.yloc = nets[i].end.yloc + 5;
                     w2-> leftend.yloc = w1-> rightend.yloc;
                     }
            else
                  {
                  w1-> rightend.yloc = w2-> rightend.yloc; /* type 3 */
                  w1-> right = w2-> right;
                  w1-> right-> left = w1;
                  }
      else
            if (w2-> right-> rightend.yloc == nets[i].end.yloc) /* type 2 */
                  {
                  w2-> rightend.yloc = nets[i].end.yloc;
                  w2-> right = NULL;
                  }
            else
                  {                       /* type 4 */
                  w2-> rightend.yloc = w2-> right-> rightend.yloc;
                  w2-> right = w2-> right-> right;
                  w2-> right-> left = w2;
                  }
else   /* vertical channel */
```

```
        if (w1-> rightend.xloc == c-> center)       /* type 1 or 3 */
                if (w2-> rightend.xloc == nets[i].end.xloc)        /* type 1 */
                        if (nets[i].layer[0] == nets[i].layer[1])
                          {
                          w1-> rightend.xloc = nets[i].end.xloc;
                          w1-> right = NULL;
                          }
                        else
                         if (nets[i].start.xloc <  nets[i].end.xloc)
                           {
                           w1-> rightend.xloc = nets[i].end.xloc - 5;
                           w2-> leftend.xloc = w1-> rightend.xloc;
                           }
                         else
                           {
                           w1-> rightend.xloc = nets[i].end.xloc + 5;
                           w2-> leftend.xloc = w1-> rightend.xloc;
                           }
                else
                        {
                        w1-> rightend.xloc = w2-> rightend.xloc;  /* type 3 */
                        w1-> right = w2-> right;
                        w1-> right-> left = w1;
                        }
        else
                if (w2-> right-> rightend.xloc == nets[i].end.xloc)  /* type 2 */
                        {
                        w2-> rightend.xloc = nets[i].end.xloc;
                        w2-> right = NULL;
                        }
                else
                        {                                   /* type 4 */
                        w2-> rightend.xloc = w2-> right-> rightend.xloc;
                        w2-> right = w2-> right-> right;
                        w2-> right-> left = w2;
                        }
}

printy()
/***************************************************************************

FUNCTION:

PURPOSE:

***************************************************************************/
{
int     i;
struct  wireseg       *pt;

if (mmetal)
        printf("metal available\n");
if (mmetal2)
        printf("metal2 available\n");
```

```
if (ppoly)
        printf("poly available\n");
if (ppoly2)
        printf("poly2 available\n");
if (ddiff)
        printf("diff available\n");
printf("\n");

for (i=0; i< lhchan; i++)
        printf("hchan %d corner %d %d opcorner %d %d\n",
        i, hchan[i].corner.xloc, hchan[i].corner.yloc,
        hchan[i].opcorner.xloc, hchan[i].opcorner.yloc);
printf("\n");

for (i=0; i< lvchan; i++)
        printf("vchan %d corner %d %d opcorner %d %d\n",
        i, vchan[i].corner.xloc, vchan[i].corner.yloc,
        vchan[i].opcorner.xloc, vchan[i].opcorner.yloc);
printf("\n");

for (i=0; i< lnet; i++)
        {
        printf("net %d from %d %d %c to %d %d %c \n",
        i, nets[i].start.xloc, nets[i].start.yloc, nets[i].layer[0],
        nets[i].end.xloc, nets[i].end.yloc, nets[i].layer[1]);

        printf("start at channel with center at %d\n",
        nets[i].pointer[0]-> center);

        printf("end at channel with center at %d\n\n",
        nets[i].pointer[1]-> center);
        }

for (i=0; i < lnet; i++)
        {
        printf("net %d\n",i);

        pt = nets[i].wpoint;
        while (pt != NULL)
                {
                printf("left %d,%d  right %d,%d\n",
                pt-> leftend.xloc, pt-> leftend.yloc,
                pt-> rightend.xloc, pt-> rightend.yloc);

                pt = pt-> right;
                }
        }
}

center_start()
/**************************************************************************

FUNCTION:    center_start
```

PURPOSE: The purpose of this routine is to build a path from the starting end point to the center of the channel that it is in. If the end point starts on a horizontal channel and begins on the metal layer the path must go to the metal2 layer first. If the end point starts on a vertical channel and begins on the metal2 layer the path must go to the metal layer first.

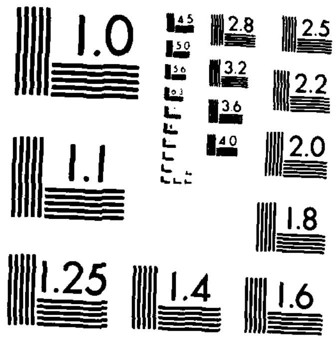
```
************************************************************************/
{
int     i;
struct  channel       *c;
struct  wireseg *w1, *w2;

for (i=0; i < lnet; i++)        /* for all nets build path to center */
      {
      c = nets[i].pointer[0];         /* c contains the starting channel addr */

      w1 = &(c-> untrack[(c-> luseg)]); /* w1 1st available wireseg addr */
      (c-> luseg)++;

      w2 = &(c-> untrack[(c-> luseg)]); /* w2 2nd available wireseg addr */

      nets[i].wpoint = w1; /* addr of the start of the net */

      if (c-> id == 1)                /* if vertical channel */

            if (nets[i].layer[0] != '2')      /* if layer != metal2 */
              {
              w1->leftend.xloc = nets[i].start.xloc;
              w1->leftend.yloc = nets[i].start.yloc;  /* go straight to */
              w1->rightend.xloc = c-> center;    /* center of chan */
              w1->rightend.yloc = w1-> leftend.yloc;
              }
            else            /* layer = metal2 */
              {
              w1->leftend.xloc = nets[i].start.xloc;  /* go to metal and */
              w1->leftend.yloc = nets[i].start.yloc;  /* then to center  */
              w1->rightend.yloc = nets[i].start.yloc;
              if (nets[i].start.xloc < c-> center)
                   w1-> rightend.xloc = nets[i].start.xloc + 5;
              else
                   w1-> rightend.xloc = nets[i].start.xloc - 5;
              w1->right = w2;
              w2->left = w1;
              w2->leftend.xloc = w1-> rightend.xloc;
              w2->leftend.yloc = w1-> rightend.yloc;
              w2->rightend.xloc = c-> center;
              w2->rightend.yloc = w2-> leftend.yloc;
              (c-> luseg)++;
              }
      else
            if (nets[i].layer[0] != 'm')   /* if layer != metal */
              {
              w1->leftend.xloc = nets[i].start.xloc;
```

```
            w1->leftend.yloc = nets[i].start.yloc;  /* go straight to */
            w1->rightend.xloc = nets[i].start.xloc; /* the center     */
            w1->rightend.yloc = c->center;
            }
          else                          /* layer = metal */
            {
            w1->leftend.xloc = nets[i].start.xloc;
            w1->leftend.yloc = nets[i].start.yloc;
            w1->rightend.xloc = nets[i].start.xloc; /* go to metal2  */
            if (nets[i].start.yloc < c->center)     /* then center   */
                 w1->rightend.yloc = nets[i].start.yloc + 5;
            else
                 w1->rightend.yloc = nets[i].start.yloc - 5;
            w1->right = w2;
            w2->left = w1;
            w2->leftend.xloc = w1->rightend.xloc;
            w2->leftend.yloc = w1->rightend.yloc;
            w2->rightend.yloc = c->center;
            w2->rightend.xloc = w2->leftend.xloc;
            (c->luseg)++;
            }
      }
}

center_end()
/***********************************************************************

FUNCTION:   center_end

PURPOSE:    The purpose of this routine is to build a path from the
      ending end point to the center of the channel that it is in.
      If the end point starts on a horizontal channel and begins on
      the metal layer the path must go to the metal2 layer first.  If
      the end point starts on a vertical channel and begins on the
      metal2 layer the path must go to the metal layer first.

***********************************************************************/
{
int     i;
struct  channel     *c;
struct  wireseg *w1, *w2, *w3;

for (i=0; i < lnet; i++)        /* for all nets process ending endpoint */
      {
      c = nets[i].pointer[1]; /* c is the channel the endpoint is in  */
      w1 = &(c->untrack[(c->luseg)]);  /* w1 is 1st unused wire seg   */
      (c->luseg)++;
      w2 = &(c->untrack[(c->luseg)]);  /* w2 is 2nd unused wire seg   */

      w3 = nets[i].wpoint;             /* w3 is 1st wire seg of net   */
      while (w3->right != NULL)
            w3 = w3->right;            /* find the end of net chain   */
      w3->right = w1;
```

```
if (c-> id == 1)      /* if channel is vertical */
       if (nets[i].layer[1] != '2')              /* if layer != metal2 */
         {
         w1-> rightend.xloc = nets[i].end.xloc;
         w1-> rightend.yloc = nets[i].end.yloc; /* go straight to    */
         w1-> leftend.xloc = c-> center;    /* the center of the */
         w1-> leftend.yloc = w1-> rightend.yloc; /* channel          */
         }
       else        /* layer = metal2 */
         {
         w1-> rightend.xloc = nets[i].end.xloc; /* go to metal and  */
         w1-> rightend.yloc = nets[i].end.yloc; /* then go to the   */
         w1-> leftend.yloc = nets[i].end.yloc;  /* center          */
         if (nets[i].end.xloc < c-> center)
              w1-> leftend.xloc = nets[i].end.xloc + 5;
         else
              w1-> leftend.xloc = nets[i].end.xloc - 5;
         w1-> left = w2;
         w2-> right = w1;
         w2-> rightend.xloc = w1-> leftend.xloc;
         w2-> rightend.yloc = w1-> leftend.yloc;
         w2-> leftend.xloc = c-> center;
         w2-> leftend.yloc = w2-> rightend.yloc;
         w3-> right = w2;
         (c-> luseg)++;
         }
else                                    /* endpoint is on horiz chan */
       if (nets[i].layer[1] != 'm')    /* layer != metal         */
         {
         w1-> rightend.xloc = nets[i].end.xloc;
         w1-> rightend.yloc = nets[i].end.yloc; /* go straight to  */
         w1-> leftend.xloc = nets[i].end.xloc;  /* the center      */
         w1-> leftend.yloc = c-> center;
         }
       else            /* layer = metal */
         {
         w1-> rightend.xloc = nets[i].end.xloc;
         w1-> rightend.yloc = nets[i].end.yloc; /* go to metal2     */
         w1-> leftend.xloc = nets[i].end.xloc;  /* and then center  */
         if (nets[i].end.yloc < c-> center)
              w1-> leftend.yloc = nets[i].end.yloc + 5;
         else
              w1-> leftend.yloc = nets[i].end.yloc - 5;
         w1-> left = w2;
         w2-> right = w1;
         w2-> rightend.xloc = w1-> leftend.xloc;
         w2-> rightend.yloc = w1-> leftend.yloc;
         w2-> leftend.yloc = c-> center;
         w2-> leftend.xloc = w2-> rightend.xloc;
         w3-> right = w2;
         (c-> luseg)++;
         }
  }
}
```

AD-A151 718 AUTOMATIC VLSI (VERY LARGE SCALE INTEGRATION) ROUTING 2/2
USING 2-LAYER METAL(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T G HEWITT
UNCLASSIFIED DEC 83 AFIT/GCS/EE/84S-2 F/G 20/12 NL

END
FILMED
DTIC




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963 A

```
horizontal(w1, w2, c)
struct  wireseg       *w1;
struct  wireseg *w2;
struct  channel       *c;
/*************************************************************************

FUNCTION:     horizontal

PURPOSE:      The purpose of this routine is to find a path between
        endpoints of a net. The endpoints must already be in the
        center of their respective channels. The subroutines horizontal
        and vertical are called recursively until the path is complete.
        The inputs to this routine are the two wire segments to be
        connected along with the starting channel.

*************************************************************************/
{
int     i;
int     small;
int     delta;
int     index;

struct  wireseg *w3;

if (w1->rightend.yloc == w2->leftend.yloc)      /* the two segments are in  */
        {                                       /* the same channel        */
        w3 = &(c->track[c->ltseg]);                  /* w3 is the connecting seg */
        (c->ltseg)++;

        adj_pointers(w1, w3, w2);          /* put w3 between w1 and w2 */

        w3->leftend.xloc = w1->rightend.xloc;
        w3->leftend.yloc = w1->rightend.yloc;
        w3->rightend.xloc = w2->leftend.xloc;
        w3->rightend.yloc = w2->leftend.yloc;
        }
else                                            /* two segments are in diff */
        {                                       /* channels                */
        small = 99999;
        delta = 99999;

        for (i=0; i < lvchan; i++)      /* search for closest vertical chan */
                {                       /* to w2                        */
                delta = vchan[i].center - w2->rightend.xloc;
                if (delta == 0)
                        {               /* if 0 closest channel found       */
                        index = i;
                        break;
                        }
                if (delta < 0)                  /* if not 0 then is it       */
                        delta = 0 - delta;      /* closer than closest yet */
                if (delta < small)
                        {
```

```
                    small = delta;
                    index = i;
                    }
            }
        w3 = &(c-> track[c-> ltseg]);        /* w3 is the next segment in chain */
        (c-> ltseg)++;
        c = &(vchan[index]);         /* c points to the new channel     */

        adj_pointers(w1, w3, w2);  /* put w3 between w1 and w2       */

        w3-> leftend.xloc = w1-> rightend.xloc;
        w3-> leftend.yloc = w1-> rightend.yloc;
        w3-> rightend.xloc = c-> center;
        w3-> rightend.yloc = w3-> leftend.yloc;

        vertical(w3, w2, c);         /* call vertical routine to complete */
        }                            /* the net chain                 */
}

vertical(w1, w2, c)
struct  wireseg      *w1;
struct  wireseg *w2;
struct  channel      *c;
/************************************************************************

FUNCTION:    vertical

PURPOSE:      The purpose of this routine is to find a path between
        endpoints of a net. The endpoints must already be in the
        center of their respective channels. The subroutines horizontal
        and vertical are called recursively until the path is complete.
        The inputs to this routine are the two wire segments to be
        connected along with the starting channel.

************************************************************************/
{
int     i;
int     small;
int     delta;
int     index;

struct  wireseg *w3;

if (w1-> rightend.xloc == w2-> leftend.xloc)     /* w1 and w2 are in same */
        {                                         /* channel          */
        w3 = &(c-> track[c-> ltseg]);                /* w3 is next unused seg */
        (c-> ltseg)++;

        adj_pointers(w1, w3, w2);            /* put w3 between w1 and w2 */

        w3-> leftend.xloc = w1-> rightend.xloc;
        w3-> leftend.yloc = w1-> rightend.yloc;
        w3-> rightend.xloc = w2-> leftend.xloc;
        w3-> rightend.yloc = w2-> leftend.yloc;
```

```
        }
else                                            /* w1 and w2 are in diff   */
        {                                       /* channels                */
        small = 99999;
        delta = 99999;

        for (i=0; i < lhchan; i++)      /* search for closest horiz chan   */
                {                       /* to w2                           */
                delta = hchan[i].center - w2->rightend.yloc;
                if (delta == 0)
                        {
                        index = i;      /* if 0 closest channel found      */
                        break;
                        }
                if (delta < 0)          /* if not 0 is channel closer than */
                        delta = 0 - delta;      /* the channel found yet   */
                if (delta < small)
                        {
                        small = delta;
                        index = i;
                        }
                }
        w3 = &(c->track[c->ltseg]);             /* w3 next unused wire seg         */
        (c->ltseg)++;
        c = &(hchan[index]);            /* c points to new channel         */

        adj_pointers(w1, w3, w2);  /* put w3 between w1 and w2         */

        w3->leftend.xloc = w1->rightend.xloc;
        w3->leftend.yloc = w1->rightend.yloc;
        w3->rightend.yloc = c->center;
        w3->rightend.xloc = w3->leftend.xloc;

        horizontal(w3, w2, c);                  /* call horizontal to complete the */
        }                               /* net chain                       */
}

adj_pointers(w1, w3, w2)
struct wireseg *w1, *w2, *w3;
/************************************************************************

FUNCTION:    adj_pointers

PURPOSE:     The purpose of this routine is to adjust pointers when
        w3 is to be put between w1 and w2. W1 and w2 are part of a
        chain and w3 is to be inserted between them.

************************************************************************/
{
w3->right = w1->right;
w1->right = w3;
w2->left  = w3;
w3->left  = w1;
}
```

*track.c*

```
#include       "auto.h"

assign_tracks()
/****************************************************************************

FUNCTION:    assign_tracks

PURPOSE:     The purpose of this routine is to call the routines that
        will resolve type 1 and 2 conflicts, check channel capacity,
        and finally route wire segments to specific tracks.

****************************************************************************/
{
int     i,k;            /* channel index */

sort_points(); /* for each wire segment...leftend has smallest x - horiz */
               /*                   ...leftend has smallest y - vert  */

for (i=0; i < lhchan; i++)      /* for all horizontal channels */
        {
        resolve_hconflicts(i);  /* resolve type 1 and 2 conflicts */
        check_hcapacity(i);     /* check channel capacity        */
        hroute(i);              /* route wire segments           */
        }

for (i=0; i < lvchan; i++)      /* for all vertical channels     */
        {
        resolve_yconflicts(i);  /* resolve type 1 and 2 conflicts */
      check_ycapacity(i);       /* check channel capacity        */
        vroute(i);              /* route wire segments             */
        }
}

resolve_hconflicts(i)
int     i;      /* channel index */
/****************************************************************************

FUNCTION:    resolve_hconflicts

PURPOSE:     The purpose of this routine is to resolve type 1 and 2
        conflicts. A type 1 conflict occurs when two wire segments start
        at the same x location. To resolve the wire segment on top must
        be routed first. A type 2 conflict occurs when two wire segments
        start and end at the same x locations but have swapped sides. To
        resolve a dogleg must be inserted.

****************************************************************************/
{
int     j, k, limit, temp;
struct  wireseg      *w1, *w2, *w3, *w4;
struct  channel *c;

sort_xloc(i,0);/* sort wire segments by leftend x coordinate */
```

```
c = &(hchan[i]);        /* c contains the address of the channel */
limit = c-> ltseg - 1;

for (j=0; j < limit; j++)        /* for all wire segments in channel */
        {
        w1 = &(c-> track[s_array[j]]);        /* w1 is first wire segment */
        w2 = &(c-> track[s_array[(j+1)]]); /* w2 is second wire seg */

        if (w1-> leftend.xloc == w2-> leftend.xloc)        /* conflict? */
                {
                if ((w2-> left-> leftend.yloc - c-> center > = 0) &&  /* type 1? */
                   (w2-> left-> rightend.yloc - c-> center > = 0) && /* switch  */
                   (w2-> right-> rightend.yloc - c-> center > = 0) && /* order  */
                   (w2-> right-> leftend.yloc - c-> center > = 0))
                        {
                        temp = s_array[j];
                        s_array[j] = s_array[(j+1)];  /* reverse order */
                        s_array[(j+1)] = temp;
                        }
                else
                if ((w2-> left-> leftend.yloc - c-> center < = 0) &&  /* type 1 */
                   (w2-> left-> rightend.yloc - c-> center < = 0) && /* do not */
                   (w2-> right-> rightend.yloc - c-> center < = 0) && /* switch*/
                   (w2-> right-> leftend.yloc - c-> center < = 0))            /* order */
                        NULL;
                else
                 if (w1-> rightend.xloc == w2-> rightend.xloc)  /* type 2? */
                        {

                        w3 = &(c-> track[c-> ltseg]);        /* create two new    */
                        (c-> ltseg)++;                /*    segments      */
                        w4 = &(c-> untrack[c-> luseg]);
                        (c-> luseg)++;

                        if (w2-> rightend.xloc == w2-> right-> leftend.xloc)
                                {
                                w3-> right = w2-> right;    /* adjust pointers */
                                w2-> right-> left = w3;
                                w2-> right = w3;
                                adj_pointers(w2, w4, w3);
                                }
                        else
                                {
                                w3-> left = w2-> left;        /* adjust pointers */
                                w2-> left-> right = w3;
                                w2-> left  = w4;
                                w4-> left  = w3;
                                w4-> right = w2;
                                w3-> right = w4;
                                }

                        w3-> rightend.xloc = w2-> rightend.xloc;
                        w3-> rightend.yloc = w2-> rightend.yloc;
```

```
                    temp = new_x(c, w2, j);            /* find mid point */

                    w2-> rightend.xloc = temp;
                    w4-> leftend.xloc  = temp;
                    w4-> leftend.yloc  = c-> center;   /* adjust segments */
                    w4-> rightend.xloc = temp;
                    w4-> rightend.yloc = c-> center;
                    w3-> leftend.xloc  = temp;
                    w3-> leftend.yloc  = c-> center;

                    sort_xloc(i,j);              /* resort segments */
                    limit++;

/* find w3 in array */      for (k = j; k <  c-> ltseg; k++)
                            if (&(c-> track[s_array[k]]) == w3)
                                break;

/* put w3 1st */    temp = s_array[k];
                    k--;
                    while (k >= j)
                            {
                            s_array[(k+1)] = s_array[k];
                            k--;
                            }
                    s_array[j] = temp;
                    j = j + 2;
                    }
            }
        }
}

int new_x(c, w1, j)
int      j;              /* pointer into s_array      */
struct  wireseg         *w1;  /* pointer to a wire segment */
struct  channel *c;     /* pointer to a channel      */
/****************************************************************************

FUNCTION:     new_x

PURPOSE:      The purpose of this routine is to find a location for a
        dogleg.  To simplify things a location is found that will not
        result in any additional conflicts.

****************************************************************************/
{
int      loc1, loc2, loc3;

loc1 = w1-> leftend.xloc;                /* left bound */
loc2 = w1-> rightend.xloc;               /* right bound */
loc3 = ((loc2 - loc1) / 2) + loc1 + 10;          /* potential midpoint */

while ( loc3 + 10 <= loc2)   /* while not beyond right bound */
        if ((j >= (c-> ltseg) - 2) ||
           (loc3 <= c-> track[s_array[(j+1)]].leftend.xloc - 10))
```

```
                return (loc3);
        else
                {
                loc3 = 10 + c-> track[s_array[(j+1)]].leftend.xloc;
                j++;
                }
printf("ERROR *** this segment can not be inplemented\n");
printf("        no place for dogleg in horiz chan with center at %d\n",
    c-> center);
exit(1);
}

sort_xloc(i,j)
int     i;      /* channel index */
int     j;      /* starting point of sort */
/*****************************************************************************

FUNCTION:    sort_xloc

PURPOSE:     The purpose of this routine is to sort the wire segments in
        a horizontal channel by x location. Rather than sort the wire segments
        themselves an array of pointers is sorted, s_array[]. If the wire
        segments are at the same x location, the wire segment that goes up
        at the leftend is put first. The inputs are what channel is to be
        sorted and where is the sorting to start.

*****************************************************************************/
{
int     k;
int     flag;
int     temp;
int     limit;

struct  wireseg     *w1, *w2, *wt;
struct  channel *c;

limit = hchan[i].ltseg;                 /* limit contains number of wire segments */
c = &(hchan[i]);                /* c contains address of channel        */

if (j == 0)                             /* initialize sort array */
        for (k=0; k <  TRKSEGS; k++)
                s_array[k] = k;

flag = TRUE;            /* flag = TRUE means sorting is not done */

while (flag)            /* while sorting is to be done          */
        {
        flag = FALSE;
        limit--;
        for (k=j; k <  limit; k++)      /* for all segments to be sorted */
                {
                w1 = &(hchan[i].track[s_array[k]]);        /* w1 is 1st seg  */
                w2 = &(hchan[i].track[s_array[(k+1)]]); /* w2 is next seg */
```

```
                if (w1-> leftend.xloc == w2-> leftend.xloc) /* seg are equal */
                {
                        if (w1-> leftend.xloc == w1-> left-> leftend.xloc)
                          {
                          if (w1-> left-> leftend.yloc == c-> center)
                                {
                                .if (w1-> left-> rightend.yloc < c-> center)
                                        {
/* left pointer points from left */    temp          = s_array[k];
/* and rightend lies below chan  */ s_array[k]     = s_array[(k+1)];
/* w1 points down so switch      */ s_array[(k+1)] = temp;
/* w1 and w2 positions           */    flag = TRUE;
                                        }
                                }
                        else
                           if (w1-> left-> leftend.yloc < c-> center)
                                        {
/* left pointer points from left */    temp          = s_array[k];
/* and leftend lies below chan   */ s_array[k]     = s_array[(k+1)];
/* w1 points down so switch      */ s_array[(k+1)] = temp;
/* w1 and w2 positions           */    flag = TRUE;
                                        }
                          }
                        else
                         if (w1-> right-> leftend.yloc== c-> center)
                                {
                                .if (w1-> right-> rightend.yloc < c-> center)
                                        {
/* right pointer points from left */ temp          = s_array[k];
/* and rightend lies below chan   */ s_array[k]     = s_array[(k+1)];
/* w1 points down so switch       */        s_array[(k+1)] = temp;
/* w1 and w2 positions            */  flag = TRUE;
                                        }
                                }
                        else
                           if (w1-> right-> leftend.yloc < c-> center)
                                        {
/* right pointer points from left */ temp          = s_array[k];
/* and leftend lies below chan    */ s_array[k]     = s_array[(k+1)];
/* w1 points down so switch       */        s_array[(k+1)] = temp;
/* w1 and w2 positions            */  flag = TRUE;
                                        }
                }
                if (w1-> leftend.xloc >  w2-> leftend.xloc)
                        {
                        temp = s_array[k];              /* w1 is > w2       */
                        s_array[k] = s_array[(k+1)]; /* so switch 1 and 2 */
                        s_array[(k+1)] = temp;
                        flag = TRUE;
                        }
                }
        }

}
```

```
sort_points()
/*****************************************************************************

FUNCTION:    sort_points

PURPOSE:     The purpose of this routine is to sort the points within a
        wire segment. Wire segments that lie on horizontal segments are
        sorted by x locations. The smallest x locations will be on the
        leftend. Wire segments that lie on vertical segments are sorted
        by y locations. The smallest y locations will be on the leftend.

*****************************************************************************/
{
int     i;
int     k;
int     tem1;
int     tem2;

struct  wireseg      *w1;

for (i=0; i < lhchan; i++)     /* for all horizontal channels */
        for (k=0; k < hchan[i].ltseg; k++)   /* for all wire segments */
                {
                w1 = &(hchan[i].track[k]);  /* w1 addr of wire seg   */

                if (w1-> rightend.xloc <  w1-> leftend.xloc)
                        {
                        tem1 = w1-> leftend.xloc;
                        tem2 = w1-> leftend.yloc;

/* swap sides of seg */         w1-> leftend.xloc = w1-> rightend.xloc;
                        w1-> leftend.yloc = w1-> rightend.yloc;

                        w1-> rightend.xloc = tem1;
                        w1-> rightend.yloc = tem2;
                        }
                }

for (i=0; i < lvchan; i++)     /* for all vertical channels */
        for (k=0; k < vchan[i].ltseg; k++)   /* for all wire segments */
                {
                w1 = &(vchan[i].track[k]);

                if (w1-> rightend.yloc <  w1-> leftend.yloc)
                        {
                        tem1 = w1-> leftend.xloc;
                        tem2 = w1-> leftend.yloc;

/* swap sides of seg */         w1-> leftend.xloc = w1-> rightend.xloc;
                        w1-> leftend.yloc = w1-> rightend.yloc;

                        w1-> rightend.xloc = tem1;
                        w1-> rightend.yloc = tem2;
```

```
			}
		}
}

resolve_yconflicts(i)
int	i;	/* channel index */
/****************************************************************************

FUNCTION:	resolve_yconflicts

PURPOSE:	The purpose of this routine is to resolve type 1 and 2
	conflicts. A type 1 conflict occurs when two wire segments start
	at the same y location. To resolve the wire segment on the right
	must be routed first. A type 2 conflict occurs when two wire
	segments start and end at the same y locations but have swapped sides.
	To resolve a dogleg must be inserted.

****************************************************************************/
{
int	j, k, limit, temp;
struct	wireseg	*w1, *w2, *w3, *w4;
struct	channel *c;

sort_yloc(i,0);/* sort wire segments by leftend x coordinate */

c = &(vchan[i]);	/* c contains the address of the channel */
limit = c->ltseg - 1;

for (j=0; j < limit; j++)	/* for all wire segments in the channel */
	{
	w1 = &(c->track[s_array[j]]);  /* w1 is first wire segment	*/
	w2 = &(c->track[s_array[(j+1)]]);  /* w2 is second wire seg	*/

	if (w1->leftend.yloc == w2->leftend.yloc)	/* conflict? */
		{
		if ((w2->left->leftend.xloc - c->center >= 0) && /* type 1? */
		   (w2->left->rightend.xloc - c->center >= 0) && /* switch */
		   (w2->right->rightend.xloc - c->center >= 0) && /* order */
		   (w2->right->leftend.xloc - c->center >= 0))
			{
			temp = s_array[j];
			s_array[j] = s_array[(j+1)];  /* reverse order */
			s_array[(j+1)] = temp;
			}
		else
		if ((w2->left->leftend.xloc - c->center <= 0) && /* type 1 */
		   (w2->left->rightend.xloc - c->center <= 0) && /* do not */
		   (w2->right->rightend.xloc - c->center <= 0) && /* switch*/
		   (w2->right->leftend.xloc - c->center <= 0))	/* order */
			NULL;
		else
		 if (w1->rightend.yloc == w2->rightend.yloc)  /* type 2? */
			{
```

```
                w3 = &(c-> track[c-> ltseg]);      /* create two new   */
                (c-> ltseg)++;                  /*    segments     */
                w4 = &(c-> untrack[c-> luseg]);
                (c-> luseg)++;

                if (w2-> rightend.yloc == w2-> right-> leftend.yloc)
                        {
                        w3-> right = w2-> right;   /* adjust pointers */
                        w2-> right-> left = w3;
                        w2-> right = w3;
                        adj_pointers(w2, w4, w3);
                        }
                else
                        {
                        w3-> left = w2-> left;     /* adjust pointers */
                        w2-> left-> right = w3;
                        w2-> left  = w4;
                        w4-> left  = w3;
                        w4-> right = w2;
                        w3-> right = w4;
                        }

                w3-> rightend.xloc = w2-> rightend.xloc;
                w3-> rightend.yloc = w2-> rightend.yloc;

                temp = new_y(c, w2, j);             /* find mid point */

                w2-> rightend.yloc = temp;
                w4-> leftend.yloc  = temp;
                w4-> leftend.xloc  = c-> center;    /* adjust segments */
                w4-> rightend.yloc = temp;
                w4-> rightend.xloc = c-> center;
                w3-> leftend.yloc  = temp;
                w3-> leftend.xloc  = c-> center;

                sort_yloc(i,j);                 /* resort segments */
                limit++;

/* find w3 in array */      for (k=j; k <  c-> ltseg; k++)
                        if (&(c-> track[s_array[k]]) == w3)
                                break;

/* put w3 first */  temp = s_array[k];
                k--;
                while (k > = j)
                        {
                        s_array[(k+1)] = s_array[k];
                        k--;
                        }
                s_array[j] = temp;
                j = j = 2;
                }
        }
    }
```

```
}

int new_y(c, w1, j)
int     j;              /* pointer into s_array      */
struct  wireseg        *w1;  /* pointer to a wire segment */
struct  channel *c;    /* pointer to a channel       */
/****************************************************************************

FUNCTION:     new_y

PURPOSE:      The purpose of this routine is to find a location for a
        dogleg.  To simplify things a location is found that will not
        result in any additional conflicts.

****************************************************************************/
{
int     loc1, loc2, loc3;

loc1 = w1->leftend.yloc;              /* left bound */
loc2 = w1->rightend.yloc;             /* right bound */
loc3 = ((loc2 - loc1) / 2) + loc1 + 10;              /* potential midpoint */

while ( loc3 + 10 <= loc2)  /* while not beyond right bound */
        if ((j >= (c->ltseg) - 2) ||
           (loc3 <= c->track[s_array[(j+1)]].leftend.yloc - 10))
                return (loc3);
        else
                {
                loc3 = 10 + c->track[s_array[(j+1)]].leftend.yloc;
                j++;
                }
printf("ERROR *** this segment can not be inplemented\n");
printf("        no place for dogleg in vert chan with center at %d\n",
   c->center);
exit(1);
}

sort_yloc(i,j)
int     i;      /* channel index */
int     j;      /* starting point of sort */
/****************************************************************************

FUNCTION:     sort_yloc

PURPOSE:      The purpose of this routine is to sort the wire segments in
        a vertical channel by y location.  Rather than sort the wire segments
        themselves an array of pointers is sorted, s_array[].  If the wire
        segments are at the same x location, the wire segment that goes right
        at the bottom is put first.  The inputs are what channel is to be
        sorted and where the sorting is to begin.

****************************************************************************/
{
int     k;
```

```
            }
        reset = FALSE;
        }
    }
}

hrouter(c, w1)
struct channel      *c;
struct wireseg      *w1;
/*****************************************************************************

FUNCTION:   hrouter

PURPOSE:    The purpose of this routine is to check for type 3 conflicts
    and to assign a specific track to a wire segment.  A type 3 conflict
    occurs when the segment to be routed starts or ends where another
    segment starts or ends.  If the other segment connects above this
    segment then the segment to be routed must be skipped.

*****************************************************************************/
{
int     i;
struct wireseg      *w2;

for (i=0; i < c->ltseg; i++)            /* for all segments */
    {
    w2 = &(c->track[s_array[i]]);       /* get segment for comparison */
    if ((w2->tag == 0) && (w2 != w1))   /* if seg not routed and not = */
        {
        if (w1->leftend.xloc == w2->rightend.xloc)
           if (chknpt(w1->leftend.xloc))
               if (go_uph(w2, w2->rightend.xloc, w2->rightend.yloc))
                   return;
        if (w1->rightend.xloc == w2->leftend.xloc)
           if (chknpt(w1->rightend.xloc))
               if (go_uph(w2, w2->leftend.xloc, w2->leftend.yloc))
                   return;
        if (w1->rightend.xloc == w2->rightend.xloc)
           if (chknpt(w1->rightend.xloc))
               if (go_uph(w2, w2->rightend.xloc, w2->rightend.yloc))
                   return;
        }
    }

w1->leftend.yloc  = top;    /* adjust w1 */
w1->rightend.yloc = top;

if (w1->left->leftend.yloc == c->center) /* adjust w1->left */
    w1->left->leftend.yloc = top;
else
    w1->left->rightend.yloc = top;

if (w1->right->leftend.yloc == c->center)       /* adjust w1->right */
    w1->right->leftend.yloc = top;
```

```
#include    "auto.h"

hroute(i)
int     i;      /* channel index */
/******************************************************************************

FUNCTION:   hroute

PURPOSE:    The purpose of this routine is to route horizontal channels
        and to resolve type 3 conflicts. The wire segments are routed
        starting at the right, from top to bottom.

******************************************************************************/
{
int     flag, reset, j;
struct  wireseg *w1, *w2;
struct  channel *c;

c = &(hchan[i]);        /* address of channel to be routed */

top = top - MINDIST;            /* beginning yloc of the first track */

if (c-> corner.xloc < c-> opcorner.xloc)  /* the beginning xloc of the track   */
        resetptr = c-> corner.xloc;
else
        resetptr = c-> opcorner.xloc;
trackptr = resetptr;                    /* current xloc of the track      */

flag = TRUE;
reset = FALSE;
while (flag)                    /* while there are wire segments to route    */
        {
        flag = FALSE;
        for (j=0; j < c-> ltseg; j++)  /* for all wire segments */
                {
                w1 = &(c-> track[s_array[j]]);       /* wire segment to route */
                if (w1-> tag != 1)              /* if not already routed */
                        {
                        flag = TRUE;
                        if (w1-> leftend.xloc > = trackptr)
                                hrouter(c, w1);                 /* routing routine */
                        else
                                reset = TRUE;
                        }
                }
        if (reset)      /* is this track full */
                {
                trackptr = resetptr;  /* reset track pointer */
                top = top - MINDIST;  /* get next track      */
                if (top < (bottom + MINDIST))
              {
                 printf("ERROR *** overflow type 3 conflict horiz chan\n");
                 printf("        with center at %d\n",hchan[i].center);
                 exit(1);
```

```
        w2->leftend.yloc = w1->leftend.yloc;             /* adjust leftend */
        w2->leftend.xloc = c2->center;

        w2->rightend.yloc = w1->rightend.yloc;           /* adjust rightend */
        w2->rightend.xloc = c2->center;

        w2->right = w1->right;                           /* adjust right and */
        w2->left  = w1->left;                            /* left pointer     */

        if (w2->left->leftend.xloc == c1->center)
                w2->left->leftend.xloc = c2->center;     /* adjust endpoint */
        else
                w2->left->rightend.xloc = c2->center;

        if (w2->right->leftend.xloc == c1->center)
                w2->right->leftend.xloc = c2->center;    /* adjust endpoint */
        else
                w2->right->rightend.xloc = c2->center;

        if (w2->left->left == w1)
                w2->left->left = w2;             /* adjust ptr to new seg */
        else
                w2->left->right = w2;

        if (w2->right->left == w1)
                w2->right->left = w2;            /* adjust ptr to new seg */
        else
                w2->right->right = w2;

for (i=j; i < (c1->ltseg - 1); i++)          /* remove w1        */
        c1->track[i] = c1->track[(i+1)];     /* from track array */
for (i=0; i < (c1->ltseg -1); i++)
        if (s_array[i] == j)
                break;
for (k=i; k < (c1->ltseg - 1); k++)          /* from sort array  */
        s_array[k] = s_array[(k+1)];
(c1->ltseg)--;

}
```

```
printf("ERROR *** this vert channel overflowed with center at %d\n",
    vchan[i].center);
exit(1);
}

new_ysegment (c1, j, w1)
struct channel *c1;  /* pointer to horizontal channel        */
int     j;
struct wireseg      *w1;  /* pointer to wireseg to remove        */
/*****************************************************************************

FUNCTION:    new_ysegment

PURPOSE:     The purpose of this routine is to build a new segment
       that will replace one that is being removed from a channel. The
       sight for the new segment is as close to the original as
       possible.

*****************************************************************************/
{
struct wireseg      *w2;  /* pointer to wireseg that may be created */
struct wireseg      *w3;  /* pointer to wireseg on left of w1      */
struct wireseg      *w4;  /* pointer to wireseg on right of w1      */

struct channel *c2;  /* pointer to a vertical channel        */

int     k, i, temp, best, index;

best = 999999;

for (i=0; i < lvchan; i++)    /* find closest hchan to c1 */
      {
      c2 = &(vchan[i]);
      if ((c2==c1) || (c2->done == 1))
            NULL;                  /* this channel not eligible */
      else
            {
            temp = c1->center - c2->center;   /* find diff between chan */
            if (temp < 0)
                  temp = 0 - temp;              /* if neg make pos */
            if (temp < best)
                  {
                  best = temp;
                  index = i;      /* keep if chan closer than last */
                  }
            }
      }
if (best == 999999)
      error("ERROR ***","cant find another vertical channel for alt path");

c2 = &(vchan[index]);
w2 = &(c2->track[c2->ltseg]);
(c2->ltseg)++;
```

```
                    }                                   /* channel      */
                else
                    {
                    tchan1 = c2->opcorner.yloc;
                    bchan1 = c2->corner.yloc;
                    }

                if ((w1->leftend.yloc < tchan1) &&
                   (w1->leftend.yloc > bchan1))
                    for (l=0; l < lhchan; l++)
                        {
                        c3 = &(hchan[l]);
                        if (c3->corner.yloc > c3->opcorner.yloc)
                          {
                          tchan1 = c3->corner.yloc;
                          bchan1 = c3->opcorner.yloc;
/* find top and bottom of */  }
/* channel              */    else
                          {
                          tchan1 = c3->opcorner.yloc;
                          bchan1 = c3->corner.yloc;
                          }
                        if ((w1->rightend.yloc < tchan1) &&
                          (w1->rightend.yloc > bchan1))
                            {
/* leftend in channel and */    new_ysegment(c1, j, w1);
/* also rightend         */     return;
                            }
                        }
                else
                 if ((w1->rightend.yloc < tchan1)  &&
                    (w1->rightend.yloc > bchan1))
                    for (l=0; l < lhchan; l++)
                        {
                        c3 = &(hchan[l]);
                        if (c3->corner.yloc > c3->opcorner.yloc)
                          {
                          tchan1 = c3->corner.yloc;
                          bchan1 = c3->opcorner.yloc;
                          }
/* find top and bottom of */ else
/* the channel          */    {
                          tchan1 = c3->opcorner.yloc;
                          bchan1 = c3->corner.yloc;
                          }
                        if ((w1->leftend.yloc < tchan1) &&
                          (w1->leftend.yloc > bchan1))
                            {
/* rightend in channel and */       new_ysegment(c1, j, w1);
/* also leftend          */     return;
                            }
                        }
                }
            }
```

```
                (w1->leftend.yloc <= w2->rightend.yloc))
                    tracks1++;
            if (((w1->rightend.yloc + MINDIST) >= w2->leftend.yloc) &&
                ((w1->rightend.yloc + MINDIST) <= w2->rightend.yloc))
                    tracks2++;
            }
        if (tracks1 > tracks2)          /* compare > of tracks 1 and 2 */
            {                           /* with tracks already counted */
            if (tracks1 > tracks)
                tracks = tracks1;
            }
        else
            {
            if (tracks2 > tracks)
                tracks = tracks2;
            }
        }

return (tracks);        /* return number of tracks already needed */
}

alternate_ypath(i)
int     i;              /* channel index */
/*****************************************************************************

FUNCTION:    alternate_ypath

PURPOSE:     The purpose of this routine is to re route a net so that it
        does not need to go through the channel specified. Segments that
        will be moved must have their endpoints in horizontal channels.
        In this way only one new segment must be found and not a new path.

*****************************************************************************/
{
struct  wireseg         *w1;    /* pointer to wireseg that may be removed */

struct  channel *c1;    /* pointer to a vertical channel        */
struct  channel *c2;    /* pointer to a horizontal channel      */
struct  channel         *c3;    /* pointer to a horizontal channel      */

int     flag, j, k, l, tchan1, bchan1;

c1 = &(vchan[i]);       /* c1 addr of channel to be reduced             */

for (j=0; j < c1->ltseg; j++) /* for all segments in channel c1 */
        {
        w1 = &(c1->track[j]);
        for (k=0; k < lhchan; k++)  /* for all horizontal channels */
            {
            c2 = &(hchan[k]);       /* addr of horiz chan       */
            if (c2->corner.yloc > c2->opcorner.yloc)
                {
                tchan1 = c2->corner.yloc;  /* find top and */
                bchan1 = c2->opcorner.yloc;        /* bottom of    */
```

```
top1   = top;
for (;;)        /* count tracks in channel until done */
        {
        top1 = top1 - MINDIST;
        if ((top1 - MINDIST) > bottom)
                tracks++;               /* count tracks available */
        else
                break;
        }

tracksn = vtracks_needed(i); /* how many tracks are needed ? */

if (tracks < tracksn)           /* if not enough tracks */
        {
        printf("ERROR *** this vertical channel overflowed with\n");
        printf("        center at %d\n",vchan[i].center);
        exit(1);
        }

c-> done = TRUE;                        /* this channel is ready to route */
}

int     vtracks_needed(i)
int     i;      /* channel index */
/***************************************************************************

FUNCTION:     vtracks_needed

PURPOSE:      The purpose of this routine is to figure out how many tracks
        are needed to route a channel. A count is made for each wire segment.
        The count is incremented each time another segment includes an end
        point of the current segment. The maximum count for any segment is
        the tracks needed for that channel.

***************************************************************************/
{
int     j, k, tracks1, tracks2, tracks;

struct  wireseg       *w1, *w2;
struct  channel *c;

tracks = 0;
c = &(vchan[i]);        /* c contains the address for the channel */

for (j=0; j < c-> ltseg; j++)  /* for all wire segments */
        {
        w1 = &(c-> track[j]);          /* wire seg to compare against     */
        tracks1 = 0;            /* tracks1 = count of seg for left */
        tracks2 = 0;            /* tracks2 count of seg for right  */

        for (k=0; k < c-> ltseg; k++)          /* for all wire segments */
                {
                w2 = &(c-> track[k]);          /* seg to compare with w1 */
                if ((w1-> leftend.yloc > = w2-> leftend.yloc) &&
```

```
for (i=j; i < (c1->ltseg - 1); i++)          /* remove w1       */
        c1->track[i] = c1->track[(i+1)];     /* from track array */
for (i=0; i < (c1->ltseg -1); i++)
        if (s_array[i] == j)
                break;
for (k=i; k < (c1->ltseg - 1); k++)          /* from sort array  */
        s_array[k] = s_array[(k+1)];
(c1->ltseg)--;

}

check_vcapacity(i)
int     i;      /* channel index */
/************************************************************************

FUNCTION:    check_vcapacity

PURPOSE:     The purpose of this routine is to check channel capacity.
        This is done by finding out how many tracks are available for
        routing.  Next, a routine is called to see how many tracks are
        needed.  If tracks needed exceed tracks available the program is
        halted and an error message is printed.

************************************************************************/
{
int     top1, tracks, tracksn, j;
struct  wireseg     *w1;
struct  channel *c;

c = &(vchan[i]);        /* channel pointer */

if (c->corner.xloc < c->opcorner.xloc)
        {
        bottom = c->corner.xloc;
        top = c->opcorner.xloc;             /* find the top and the        */
        }                           /* bottom of the channel */
else
        {
        bottom = c->opcorner.xloc;
        top = c->corner.xloc;
        }

for (j=0; j < c->luseg; j++)  /* check to see if channel has to be */
        {                     /* reduced due to vias               */
        w1 = &(c->untrack[j]);
        if ((w1->rightend.xloc < c->center) && (w1->rightend.xloc > bottom))
                bottom = w1->rightend.xloc;       /* adjust top and bottom */
        else
         if ((w1->rightend.xloc > c->center) && (w1->rightend.xloc < top))
                top = w1->rightend.xloc;
        }

tracks = 0;
```

```
best = 999999;

for (i=0; i < lhchan; i++)      /* find closest hchan to c1 */
        {
        c2 = &(hchan[i]);
        if ((c2==c1) || (c2-> done == 1))
                NULL;                       /* this channel not eligible */
        else
                {
                temp = c1-> center - c2-> center;   /* find diff between chan */
                if (temp < 0)
                        temp = 0 - temp;             /* if neg make pos */
                if (temp < best)
                        {
                        best = temp;
                        index = i;       /* keep if chan closer than last */
                        }
                }
        }
if (best == 999999)
  error("ERROR ***","cant find another horizontal chan for alternate path");

c2 = &(hchan[index]);
w2 = &(c2-> track[c2-> ltseg]);
(c2-> ltseg)++;

        w2-> leftend.xloc = w1-> leftend.xloc;          /* adjust leftend */
        w2-> leftend.yloc = c2-> center;

        w2-> rightend.xloc = w1-> rightend.xloc;        /* adjust rightend */
        w2-> rightend.yloc = c2-> center;

        w2-> right = w1-> right;                        /* adjust right and */
        w2-> left  = w1-> left;                         /* left pointer     */

        if (w2-> left-> leftend.yloc == c1-> center)
                w2-> left-> leftend.yloc = c2-> center;     /* adjust endpoint */
        else
                w2-> left-> rightend.yloc = c2-> center;

        if (w2-> right-> leftend.yloc == c1-> center)
                w2-> right-> leftend.yloc = c2-> center;   /* adjust endpoint */
        else
                w2-> right-> rightend.yloc = c2-> center;

        if (w2-> left-> left == w1)
                w2-> left-> left = w2;              /* adjust ptr to new seg */
        else
                w2-> left-> right = w2;

        if (w2-> right-> left == w1)
                w2-> right-> left = w2;             /* adjust ptr to new seg */
        else
                w2-> right-> right = w2;
```

```
        {
        w1 = &(c1-> track[j]);
        for (k=0; k < lvchan; k++)  /* for all vertical channels */
                {
                c2 = &(vchan[k]);
                if (w1-> leftend.xloc == c2-> center)  /* leftend in c2 */
                        for (l=0; l < lvchan; l++)
                                {
                                if (w1-> rightend.xloc == vchan[l].center)
                                        {
/* leftend in channel and also */       new_hsegment(c1, j, w1);
/* rightend                    */   return;
                                        }
                                }
                else
                 if (w1-> rightend.xloc == c2-> center)
                        for (l=0; l < lvchan; l++)
                                {
                                if (w1-> leftend.xloc == vchan[l].center)
                                        {
/* rightend in channel and also */  new_hsegment(c1, j, w1);
/* leftend                     */     return;
                                        }
                                }
                }
        }
printf("ERROR *** this horiz channel overflowed with center at %d\n",
    hchan[i].center);
exit(1);
}

new_hsegment (c1, j, w1)
struct  channel *c1;  /* pointer to horizontal channel        */
int     j;
struct  wireseg       *w1;  /* pointer to wireseg to remove         */
/***************************************************************************

FUNCTION:    new_hsegment

PURPOSE:     The purpose of this routine is to build a new segment
        that will replace one that is being removed from a channel.  The
        sight for the new segment is as close to the original as
        possible.

***************************************************************************/
{
struct  wireseg       *w2;  /* pointer to wireseg that may be created */
struct  wireseg       *w3;  /* pointer to wireseg on left of w1       */
struct  wireseg       *w4;  /* pointer to wireseg on right of w1      */

struct  channel *c2;  /* pointer to a vertical channel        */

int     k, i, temp, best, index;
```

```
for (j=0; j < c-> ltseg; j++)  /* for all wire segments */
        {
        w1 = &(c-> track[j]);          /* wire seg to compare against    */
        tracks1 = 0;            /* tracks1 = count of seg for left  */
        tracks2 = 0;            /* tracks2 = count of seg for right */

        for (k=0; k < c-> ltseg; k++)          /* for all wire segments  */
                {
                w2 = &(c-> track[k]);          /* seg to compare with w1 */
                if ((w1-> leftend.xloc >= w2-> leftend.xloc) &&
                   (w1-> leftend.xloc < = w2-> rightend.xloc))
                        tracks1++;
                if (((w1-> rightend.xloc + MINDIST) > = w2-> leftend.xloc) &&
                   ((w1-> rightend.xloc + MINDIST) < = w2-> rightend.xloc))
                        tracks2++;
                }
        if (tracks1 > tracks2)              /* compare > of tracks 1 and 2 */
                {                       /* with tracks already counted */
                if (tracks1 > tracks)
                        tracks = tracks1;
                }
        else
                {
                if (tracks2 > tracks)
                        tracks = tracks2;
                }
        }

return (tracks);        /* return number of tracks needed */
}

alternate_hpath(i)
int      i;             /* channel index */
/*****************************************************************************

FUNCTION:     alternate_hpath

PURPOSE:      The purpose of this routine is to re route a net so that it
        does not need to go through the channel specified. Segments that
        will be moved must have their endpoints in vertical channels. In
        this way only one new segment must be found and not a new path.

*****************************************************************************/
{
struct  wireseg        *w1;   /* pointer to wireseg that may be removed */

struct  channel *c1;  /* pointer to horizontal channel       */
struct  channel *c2;  /* pointer to a vertical channel       */

int      flag, j, k, l;

c1 = &(hchan[i]);

for (j=0; j < c1-> ltseg; j++) /* for all segments in this channel */
```

```
for (j=0; j < c->luseg; j++)  /* check to see if channel has to be */
        {                      /* reduced due to vias              */
        w1 = &(c->untrack[j]);
        if ((w1->rightend.yloc < c->center) && (w1->rightend.yloc > bottom))
                bottom = w1->rightend.yloc;      /* adjust top and bottom */
        else
          if ((w1->rightend.yloc > c->center) && (w1->rightend.yloc < top))
                top = w1->rightend.yloc;
        }

tracks = 0;
top1   = top;
for (;;)        /* count tracks in channel until done */
        {
        top1 = top1 - MINDIST;
        if ((top1 - MINDIST) > bottom)
                tracks++;               /* count tracks available */
        else
                break;
        }

tracksn = htracks_needed(i); /* how many tracks are needed ? */

while (tracks < tracksn)     /* while not enough tracks */
        {
        alternate_hpath(i);   /* reduce tracks needed */
        tracksn = htracks_needed(i);
        }

c->done = TRUE;                         /* this channel is ready to route */
}

int     htracks_needed(i)
int     i;      /* channel index */
/**************************************************************************

FUNCTION:    htracks_needed

PURPOSE:     The purpose of this routine to figure out how many tracks are
        needed to route a channel.  A count is made for each wire segment
        The count is incremented each time another segment includes an end
        point of the current segment.  The maximum count for any segment
        is the tracks needed for that channel.

**************************************************************************/
{
int     j, k, tracks1, tracks2, tracks;

struct  wireseg      *w1, *w2;
struct  channel *c;

tracks = 0;
c = &(hchan[i]);        /* c contains the address for the channel */
```

```
/* w1 points left so switch  */      s_array[(k+1)] = temp;
/* w1 and w2 positions        */          flag = TRUE;
                                          }
                              }
                         else
                          if (w1-> right-> leftend.xloc < c-> center)
                                          {
/* right pointer points to bot  */    temp = s_array[k];
/* and leftend lies on the left */    s_array[k] = s_array[(k+1)];
/* w1 points left so switch  */      s_array[(k+1)] = temp;
/* w1 and w2 positions        */          flag = TRUE;
                                          }
                   if (w1-> leftend.yloc >  w2-> leftend.yloc)
                           {
                           temp = s_array[k];          /* w1 > w2    */
                           s_array[k] = s_array[(k+1)]; /* switch 1 and 2 */
                           s_array[(k+1)] = temp;
                           flag = TRUE;
                           }
                   }
          }
}

check_hcapacity(i)
int     i;      /* channel index */
/****************************************************************************

FUNCTION:    check_hcapacity

PURPOSE:     The purpose of this routine is to check channel capacity.
        This is done by finding out how many tracks are available for
        routing.  Next, a routine is called to see how many tracks are
        needed. If tracks needed exceed tracks available then tracks needed
        are reduced by finding an alternate path for some net.

****************************************************************************/
{
int     top1, tracks, tracksn, j;
struct  wireseg     *w1;
struct  channel *c;

c = &(hchan[i]);       /* channel pointer */

if (c-> corner.yloc <  c-> opcorner.yloc)
        {
        bottom = c-> corner.yloc;
        top = c-> opcorner.yloc;              /* find the top and the        */
        }                              /* bottom of the channel */
else
        {
        bottom = c-> opcorner.yloc;
        top = c-> corner.yloc;
        }
```

```
int     flag;
int     temp;
int     limit;

struct  wireseg      *w1, *w2, *wt;
struct  channel *c;

limit = vchan[i].ltseg;               /* limit contains number of wire segments */
c = &(vchan[i]);              /* c contains address of channel        */

if (j == 0)                   /* initialize sort array */
        for (k=0; k < TRKSEGS; k++)
                s_array[k] = k;

flag = TRUE;           /* flag = TRUE means sorting is not done */

while (flag)           /* while sorting is to be done                */
        {
        flag = FALSE;
        limit--;
        for (k=j; k < limit; k++)      /* for all segments to be sorted */
                {
                w1 = &(vchan[i].track[s_array[k]]);        /* w1 is 1st seg  */
                w2 = &(vchan[i].track[s_array[(k+1)]]); /* w2 is next seg */

                if (w1-> leftend.yloc == w2-> leftend.yloc)
                        if (w1-> leftend.yloc == w1-> left-> leftend.yloc)
                          {
                          if (w1-> left-> leftend.xloc == c-> center)
                                {
                                if (w1-> left-> rightend.xloc <  c-> center)
                                        {
/* left pointer points from bot  */     temp = s_array[k];
/* and rightend lies on the left */     s_array[k] = s_array[(k+1)];
/* w1 points left so switch      */     s_array[(k+1)] = temp;
/* w1 and w2 positions            */         flag = TRUE;
                                        }
                                }
                          else
                           if (w1-> left-> leftend.xloc <  c-> center)
                                        {
/* left pointer points from bot  */     temp = s_array[k];
/* and leftend lies on the left  */     s_array[k] = s_array[(k+1)];
/* w1 points left so switch   */        s_array[(k+1)] = temp;
/* w1 and w2 positions            */         flag = TRUE;
                                        }
                          }
                        else
                         if (w1-> right-> leftend.xloc == c-> center)
                                {
                                if (w1-> right-> rightend.xloc <  c-> center)
                                        {
/* right pointer points to bot   */     temp = s_array[k];
/* and rightend lies on the left */     s_array[k] = s_array[(k+1)];
```

```
else
        w1-> right-> rightend.yloc = top;

trackptr = trackptr + w1-> rightend.xloc + MINDIST;  /* move track pointer */

w1-> tag = 1; /* mark this segment done */
}

chknpt(xloc)
int     xloc;
/*****************************************************************************

FUNCTION:     chknpt

PURPOSE:      The purpose of this routine is to determine if the endpoint
        of the segment lies in a vertical channel.  If it does then a
        type 3 conflict will not occur.  This routine returns the value
        FALSE if the endpoint lies in a vertical channel and TRUE if it
        does not.

*****************************************************************************/
{
struct  channel         *c;
int     i;

for (i=0; i < lvchan; i++)
        {
        c = &(vchan[i]);
        if (xloc == c-> center)
                return(FALSE);
        }

return(TRUE);
}

int     go_uph(w2, endptx, endpty)
struct  wireseg         *w2;
int     endptx, endpty;
/*****************************************************************************

FUNCTION:     go_uph

PURPOSE:      The purpose of this routine is to determine if wire segment
        w2 should be routed before w1.  This is determined by examining
        if the endpoint of w2 "goes up".  The input is the address of the
        wire segment and the x and y coordinates of the endpoint to be
        examined.

*****************************************************************************/
{
if (w2-> left-> leftend.xloc == endptx)     /* endpoint connects on left */

        if (w2-> left-> leftend.yloc == endpty)     /* leftend connects to endpt */
```

```
            if (w2-> left-> rightend.yloc >  endpty)
                  return(TRUE);
            else                    /* does it go up? */
                  return(FALSE);

      else                          /* rightend connects to endpt */

            if (w2-> left-> leftend.yloc >  endpty)
                  return(TRUE);
            else                    /* does it go up? */
                  return(FALSE);

else                                /* endpoint connects on right */

      if (w2-> right-> leftend.yloc == endpty) /* leftend connects on endpt */

            if (w2-> right-> rightend.yloc >  endpty)
                  return(TRUE);
            else                    /* does it go up? */
                  return(FALSE);

      else                          /* rightend connects to endpt */

            if (w2-> right-> leftend.yloc >  endpty)
                  return(TRUE);
            else                    /* does it go up? */
                  return(FALSE);
}

vroute(i)
int     i;      /* channel index */
/*****************************************************************************

FUNCTION:    vroute

PURPOSE:     The purpose of this routine is to route vertical channels
      and to resolve type 3 conflicts. The wire segments are routed
      starting from the bottom, from right to left.

*****************************************************************************/
{
int     flag, j, reset;
struct  wireseg *w1, *w2;
struct  channel *c;

c = &(vchan[i]);        /* address of channel to be routed */

top = top - MINDIST; /* beginning xloc of the first track */

if (c-> corner.yloc <  c-> opcorner.yloc) /* the beginning yloc of the track  */
      resetptr = c-> corner.yloc;
else
      resetptr = c-> opcorner.yloc;
trackptr = resetptr;                  /* current yloc of the track      */
```

```
flag = TRUE;
reset = FALSE;
while (flag)                    /* while there are wire segments to route    */
        {
        flag = FALSE;
        for (j=0; j < c-> ltseg; j++)  /* for all wire segments */
                {
                w1 = &(c-> track[s_array[j]]);      /* wire segment to route */
                if (w1-> tag != 1)             /* if not already routed */
                        {
                        flag = TRUE;
                        if (w1-> leftend.yloc > = trackptr)
                                vrouter(c, w1);         /* routing routine   */
                        else
                                reset = TRUE;
                        }
                }
        if (reset)      /* is this track full */
                {
                trackptr = resetptr;   /* reset track pointer */
                top = top - MINDIST;  /* get next track      */
                if (top < (bottom + MINDIST))
              {
                 printf("ERROR *** overflow type 3 conflict vert chan\n");
                 printf("          with center at %d\n",vchan[i].center);
                 exit(1);
                 }
                }
        }
}

vrouter(c, w1)
struct channel      *c;
struct wireseg      *w1;
/*******************************************************************************

FUNCTION:    vrouter

PURPOSE:     The purpose of this routine is to check for type 3 conflicts
        and to assign a specific track to a wire segment.  A type 3 conflict
        occurs when the segment to be routed starts and ends where another
        segment starts or ends.  If the other segment connects above this
        segment the segment to be routed must be skipped.

*******************************************************************************/
{
int     i;
struct wireseg      *w2;

for (i=0; i < c-> ltseg; i++)           /* for all segments  */
        {
        w2 = &(c-> track[s_array[i]]);       /* get segment for comparison */
        if ((w2-> tag == 0) && (w2 != w1))   /* if seg not routed and not = */
```

```
			{
			if (w1-> leftend.yloc == w2-> rightend.yloc)
				if (go_upv(w2, w2-> rightend.xloc, w2-> rightend.yloc))
					return;
			if (w1-> rightend.yloc == w2-> leftend.yloc)
				if (go_upv(w2, w2-> leftend.xloc, w2-> leftend.yloc))
					return;
			if (w1-> rightend.yloc == w2-> rightend.yloc)
				if (go_upv(w2, w2-> rightend.xloc, w2-> rightend.yloc))
					return;
			}
		}

w1-> leftend.xloc  = top;     /* adjust w1 */
w1-> rightend.xloc = top;

if (w1-> left-> leftend.xloc == c-> center) /* adjust w1-> left */
	w1-> left-> leftend.xloc = top;
else
	w1-> left-> rightend.xloc = top;

if (w1-> right-> leftend.xloc == c-> center)        /* adjust w1-> right */
	w1-> right-> leftend.xloc = top;
else
	w1-> right-> rightend.xloc = top;

trackptr = trackptr + w1-> rightend.yloc + MINDIST;  /* move track pointer */

w1-> tag = 1; /* mark this segment done */
}

int     go_upv(w2, endptx, endpty)
struct  wireseg       *w2;
int     endptx, endpty;
/*******************************************************************************

FUNCTION:    go_upv

PURPOSE:      The purpose of this routine is to determine if wire segment
	w2 should be routed before w1. This is determined by examining
	if the endpoint of w2 "goes up". The input is the address of the
	wire segment and the x and y coordinates of the endpoint to be
	examined.

*******************************************************************************/
{
if (w2-> left-> leftend.yloc == endpty)     /* endpoint connects on left */

	if (w2-> left-> leftend.xloc == endptx)     /* leftend connect to endpt */

		if (w2-> left-> rightend.xloc > endptx)
			return(TRUE);
		else                        /* does it go up? */
```

```
                return(FALSE);

        else                            /* rightend connects to endpt */

            if (w2-> left-> leftend.xloc > endptx)
                return(TRUE);
            else                        /* does it go up? */
                return(FALSE);

    else                                /* endpoint connects on right */

        if (w2-> right-> leftend.xloc == endptx)  /* leftend connects to endpt */

            if (w2-> right-> rightend.xloc > endptx)
                return(TRUE);
            else                        /* does it go up? */
                return(FALSE);

        else                            /* rightend connect to endpt */

            if (w2-> right-> leftend.xloc > endptx)
                return(TRUE);
            else                        /* does it go up? */
                return(FALSE);
}
```

*formcll.c*

```
#include      "auto.h"

form_cll()
/*************************************************************************

FUNCTION:    form_cll

PURPOSE:     The purpose of this routine is to form the output into
        CLL statements. A comment is created first describing the net.
        The CLL wire and via statements follow.

*************************************************************************/
{
int      i;       /* net index */

for (i=0; i < lnet; i++)         /* for all nets */
        {
        comment(i);              /* form comment line */

        cll(i);                  /* form CLL wire and via statements */
        }
}

comment(i)
int      i;       /* net index */
/*************************************************************************

FUNCTION:    comment

PURPOSE:     The purpose of this routine is to form a comment statement
        that will describe a net.

*************************************************************************/
{
char     layer1[10],layer2[10];

switch (nets[i].layer[0])                  /* layer1 is starting layer */
        {
        case 'm': strcpy(layer1,"metal");
                break;
        case '2': strcpy(layer1,"metal2");
                break;
        case 'p': strcpy(layer1,"poly");
                break;
        case 'P': strcpy(layer1,"poly2");
                break;
        case 'd': strcpy(layer1,"diff");
        }

switch (nets[i].layer[1])                  /* layer2 is ending layer */
        {
        case 'm': strcpy(layer2,"metal");
                break;
        case '2': strcpy(layer2,"metal2");
```

```
                break;
        case 'p': strcpy(layer2,"poly");
                break;
        case 'P': strcpy(layer2,"poly2");
                break;
        case 'd': strcpy(layer2,"diff");
        }

printf("\n/* CONNECT %d,%d %s to %d,%d %s */\n",
        nets[i].start.xloc,nets[i].start.yloc,layer1,
        nets[i].end.xloc,nets[i].end.yloc,layer2);    /* the comment */
}

cll(i)
int     i;      /* net index */
/****************************************************************************

FUNCTION:   cll

PURPOSE:     The purpose of this routine is to form CLL wire and via
        statements that will describe the route of the net.

****************************************************************************/
{
struct  wireseg       *w1;
char    layer1[10];
int     vxloc, vyloc;

w1 = nets[i].wpoint; /* address of first wire segment */

switch (nets[i].layer[0])     /* layer1 is the starting layer */
        {
        case 'm': strcpy(layer1,"metal");
                break;
        case '2': strcpy(layer1,"metal2");
                break;
        case 'p': strcpy(layer1,"poly");
                break;
        case 'P': strcpy(layer1,"poly2");
                break;
        case 'd': strcpy(layer1,"diff");
        }
printf("wire  %s   %d,%d    %d,%d;\n",layer1,
w1-> leftend.xloc,w1-> leftend.yloc,
w1-> rightend.xloc,w1-> rightend.yloc); /* first wire statement */

if (w1-> right == NULL)            /* one segment for this net */
        return;

vxloc = w1-> rightend.xloc;        /* xloc of via */
vyloc = w1-> rightend.yloc;/* yloc of via */
printf("via  %d,%d;\n",vxloc-2,vyloc-2);          /* via statement */

while (w1 != NULL)  /* while there is a segment */
```

```
{
w1 = w1-> right;     /* get next segment */

if (w1-> right == NULL)       /* is this last segment */
        {
        switch (nets[i].layer[1])      /* layer1 is ending layer */
                {
                case 'm': strcpy(layer1,"metal");
                          break;
                case '2': strcpy(layer1,"metal2");
                          break;
                case 'p': strcpy(layer1,"poly");
                          break;
                case 'P': strcpy(layer1,"poly2");
                          break;
                case 'd': strcpy(layer1,"diff");
                }
        printf("wire  %s    %d,%d    %d,%d;\n\n",layer1,
        w1-> leftend.xloc,w1-> leftend.yloc,
        w1-> rightend.xloc,w1-> rightend.yloc); /* wire statement */
        w1 = w1-> right;
        }
else            /* not the last segment */
 {
 if (w1-> leftend.xloc == w1-> rightend.xloc)
        strcpy(layer1,"metal2");       /* vertical on metal2 */
 else
        strcpy(layer1,"metal");                /* horizontal on metal */

 printf("wire  %s    %d,%d    %d,%d;\n",layer1,
 w1-> leftend.xloc,w1-> leftend.yloc,
 w1-> rightend.xloc,w1-> rightend.yloc);        /* wire statement */

 if (vxloc == w1-> leftend.xloc)     /* locate via position */
        if (vyloc == w1-> leftend.yloc)
                {
                vxloc = w1-> rightend.xloc;
                vyloc = w1-> rightend.yloc;
                }
        else
                {
                vxloc = w1-> leftend.xloc;
                vyloc = w1-> leftend.yloc;
                }
 else
        if (vyloc == w1-> rightend.yloc)
                {
                vxloc = w1-> leftend.xloc;
                vyloc = w1-> leftend.yloc;
                }
        else
                {
                vxloc = w1-> rightend.xloc;
                vyloc = w1-> rightend.yloc;
```

```
                }
        printf("via %d,%d;\n",vxloc-2,vyloc-2); /* via statement */
        }
      }
}
```

## Appendix E

## User's Manual

The automatic routing program creates CLL WIRE and VIA statements. These statements describe the routing path of two point nets. The output from the routing program is merged with CLL statements that place library cells on a grid. The new program file can be plotted and used to create VLSI chips.

### The Input File

The automatic routing program is written in C. Input is introduced to the system using standard input. That is, using the form:

```
auto < input
```

The input file contains three types of input: 1) layer input, 2) channel input, and 3) net input. The routing program uses spaces, commas, tab characters, and end-of-line characters to seperate words in the input file. Any combination of these characters can be used to format the input.

Layer input. Layer input lets the user specify what routing layers are available. If layer input is not specified, routing will be limited to four layers: 1) metal, 2) metal2, 3) poly, and 4) diff. Besides the four routing

layers named above, poly2 can also be used. When layer input is specified it must come before net input. Since most of the routing will be done on the metal and metal2 layers, both must be specified. The layer input is specified using a statement of the form:

```
BEGIN_LAYER metal, metal2, poly, diff END_LAYER
```

Channel input. Channel input lets the user specify the channels between library cells. Channels are rectangular and can be horizontal or vertical. Horizontal channels can have net endpoints above and below the channel. Vertical channels can have net endpoints on the left and right of the channel.

Each horizontal channel must intersect every vertical channel and vice versa. This limits the chip design to a matrix type organization. A channel is described by two corner points. Either the top-left and bottom-right or bottom-left and top-right corner points must be specified. Horizontal channels are specified using a statement of the form:

```
BEGIN_HCHANNELS    0,0      200,100
                   0,200    200,400 END_CHANNELS
```

Vertical channels are specified using a statement of the form:

```
BEGIN_VCHANNELS    0,0      100,500
                   100,200 200,500 END_CHANNELS
```

Net input. Net input lets the user specify two point nets. The net endpoints must lie on or outside channel boundaries. If the endpoint lies outside the channel boundary it must be closer to its target channel than any other channel. The endpoint will be routed from the closest channel. Endpoints are specified by x and y coordinates and a layer designator. A net is specified by two endpoints. Nets are shown using a statement of the form;

```
CONNECT   23,45,poly      45,87,diff
```

## The Output File

The output from the automatic routing program can go to the terminal or to an output file. To specify an output file use a statement of the form:

```
auto < input > output
```

The output contains CLL WIRE and VIA statements that describe the routing path of all nets. Each routing path is preceded by a comment stating the source and destination endpoints. Output follows this format:

```
/* CONNECT 436,500 diff to 315,0 diff */
wire  diff    436,500     436,486;
via  434,484;
wire  metal    315,486     436,486;
via  313,484;
wire  diff    315,486     315,0;
```

Plotting the Output

To obtain a plot, changes have to be made to the output file. The local CLL program does not recognize the metal2 routing layer. All references to that layer must be changed. Also, a layer designator must be added to all via statements. The output file must look like a C subroutine, that is it must be surrounded by braces and be named. The output should resemble the statements below:

```
    sample
          [
          poly;    /* global layer designator for VIA
statements */
          /* CONNECT 436,500 diff to 315,0 diff */
          wire  diff    436,500     436,486;
          via  434,484;
          wire  metal    315,486     436,486;
          via  313,484;
          wire  diff    315,486     315,0;
          ]
```

To get a plot of the routing paths the CLL program is invoked using a statement of the form:

```
          cll  output.cll
```

Note that the output file must be a .cll file.

Compiling the Program

The automatic routing program resides in six files: 1)

auto.h, 2) ainit.c, 3) atrack.c, 4) aroute.c, 5) aformcll.c, and 6) formcll.c. To compile the program use a statement of the form:

```
cc ainit.c aroute.c atrack.c aformcll.c formcll.c
```

The file auto.h is a file of #define's and global variables. The file is included in each of the above files.

Error Handling

When an error occurs the program will halt immediately. All errors must be corrected before the program will run successfully. The error messages and a brief description follow.

ERROR *** illegal input buf

This error occurs when an unidentified word is encountered in the input file. The program was expecting BEGIN_LAYER, BEGIN_HCHANNELS, BEGIN_VCHANNELS, or CONNECT. To correct error, fix the input file.

ERROR *** missing required layers

This error occurs when metal and metal2 are not specified as layers. These are the two main routing layers and must be specified.

ERROR *** not a valid layer buf

This error occurs when a net specifies an illegal layer. To correct problem, change layers.

ERROR *** this segment can not be implemented no place for dogleg in horiz chan with center at #

ERROR *** this segment can not be implemented no place for dogleg in vert chan with center at #

These errors occur while trying to resolve Type 2 conflicts. A dogleg can not be implemented without causing a new conflict in the channel. To correct the problem, space the endpoints of the nets further apart.

ERROR *** this horiz channel overflowed with center at #

ERROR *** this vert channel overflowed with center at #

This error occurs when tracks needed exceeds tracks available in a channel. For horizontal channels none of the wire segments met the removal criteria. To correct, the channel height must be increased.

ERROR *** can not find another horizontal channel for alternate path

This error occurs when all horizontal channels have been routed except for the one that overflowed. To correct the problem, change the order of routing by varying the channel input.

ERROR *** overflow type 3 conflict horiz chan with center at #

ERROR *** overflow type 3 conflict vert chan with center at #

This error occurs when tracks needed exceed tracks available because of type 3 conflicts. To correct the problem, the channel height must be increased or nets have to be removed from the channel.

## Vita

Terry Glenn Hewitt was born on 31 January 1957 in Alamo, Georgia. He graduated from high school in Alamogordo, New Mexico in 1974 and attended the University of Southern Mississippi from which he received the degree of Bachelor of Science in Computer Science in May 1978. Upon graduation, he received a commission in the USAF through the ROTC program. He served as a computer systems analyst for the 552 AWACW at Tinker AFB, Oklahoma until entering the School of Engineering, Air Force Institute of Technology, in June 1982.

Permanent Address: 206 N Montclair Ave

Brandon, Florida 33511

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| 2a. SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited. |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/84S-2 | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG | 7a. NAME OF MONITORING ORGANIZATION |
| 6c. ADDRESS (City, State and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433 | 7b. ADDRESS (City, State and ZIP Code) |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c. ADDRESS (City, State and ZIP Code) | 10. SOURCE OF FUNDING NOS. |
| 11. TITLE (Include Security Classification): See Box 19 (UNCLASSIFIED) | |

| 10. SOURCE OF FUNDING NOS. | | | |
|---|---|---|---|
| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT NO. |
| | | | |

12. PERSONAL AUTHOR(S): Terry G. Hewitt, B.S., Capt, USAF

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Yr., Mo., Day) | 15. PAGE COUNT |
|---|---|---|---|
| MS Thesis | FROM ______ TO ______ | 1983 December | 130 |

16. SUPPLEMENTARY NOTATION

| 17. COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB. GR. | Automatic Routing, VLSI Design |
| 9 | 2 | | "Dogleg" Channel Routing |

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Approved for public release: IAW AFR 190-17.
LYNN E. WOLAVER 25 Feb 85
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

Title: AUTOMATIC VLSI ROUTING USING 2-LAYER METAL

Thesis Chairman: Harold Carter, LTC, USAF


Automatic routing of a computer chip is a complex task. When routing and VLSI (Very Large Scale Integration) design are combined, the problem is increased.

A computer program was developed to automatically route the interconnections of a VLSI chip. Only two point nets can be routed using a "dogleg" channel router on both horizontal and vertical channels. The program runs very quickly. Fifty nets were routed in less than 1 second.


| | | |
|---|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐ | 21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL: Harold Carter, LTC, USAF | 22b. TELEPHONE NUMBER (Include Area Code) | 22c. OFFICE SYMBOL: AFIT/ENG |

DD FORM 1473, 83 APR    EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

The program minimizes the channel height of a channel. The channels must be rectangular. Also, each horizontal channel must intersect every vertical channel and vice versa.

Alternate paths can be found for nets in horizontal channels when channel capacity is exceeded. Constraint loops are removed by ordering the way nets are routed or by introducing a "dogleg".

The program produces output that is compatible with CLL (Chip Layout Language). The output from the program can be merged with CLL statements that place cells from a library on a grid to form plots or to create CIF (Caltech Intermediate Form) data to be used in making VLSI chips.

END

FILMED

5-85

DTIC